

# Study guide: Introduction to Finite Element Methods

Hans Petter Langtangen<sup>1,2</sup>

Center for Biomedical Computing, Simula Research Laboratory<sup>1</sup>

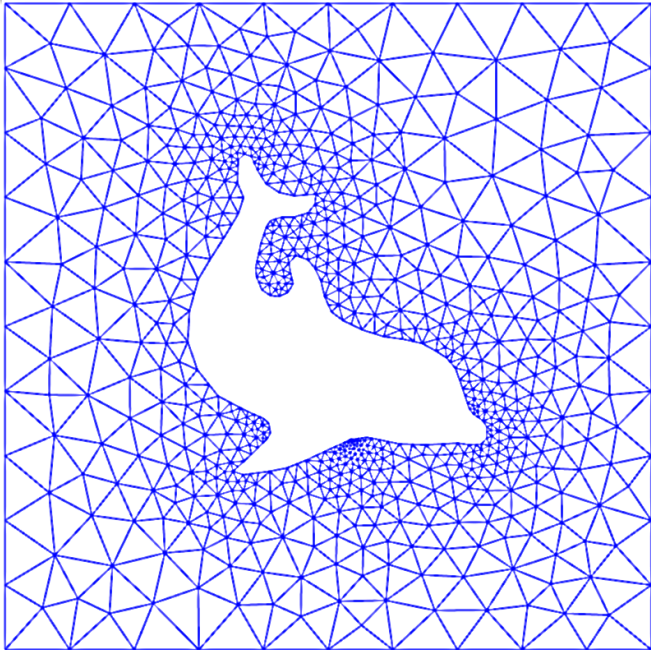
Department of Informatics, University of Oslo<sup>2</sup>

Sep 4, 2014

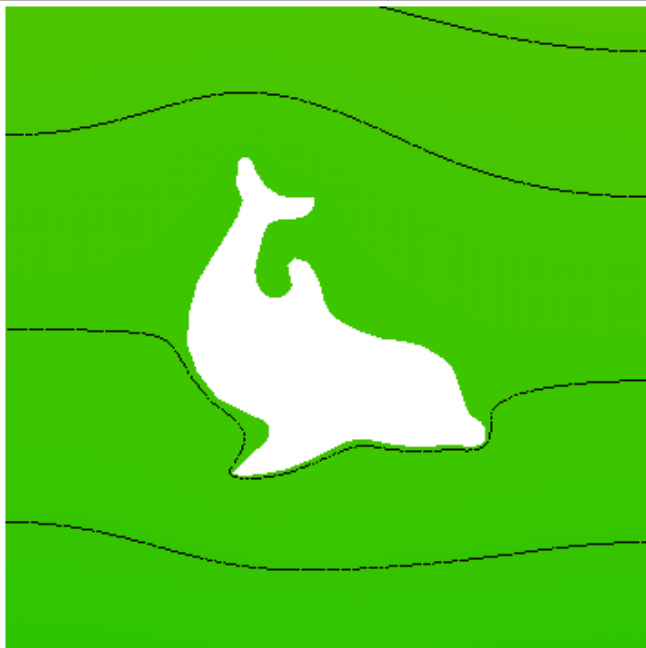
# Why finite elements?

- Can with ease solve PDEs in domains with *complex geometry*
- Can with ease provide higher-order approximations
- Has (in simpler stationary problems) a rigorous mathematical analysis framework (not much considered here)

# Domain for flow around a dolphin



## The flow



# Solving PDEs by the finite element method

## Stationary PDEs:

- Transform the PDE problem to a *variational form*
- Define function approximation over *finite elements*
- Use a machinery to derive *linear systems*
- Solve linear systems

## Time-dependent PDEs:

- Finite elements *in space*
- Finite difference (or ODE solver) in time

# We start with function approximation, then we treat PDEs

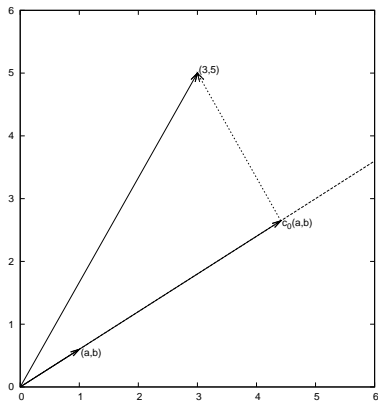
## Learning strategy.

- Start with approximation of functions, not PDEs
- Introduce finite element *approximations*
- See later how this machinery is applied to PDEs

## Reason:

The finite element method has many concepts and a jungle of details. This learning strategy minimizes the mixing of ideas, concepts, and technical details.

# Approximation in vector spaces



# Approximation set-up

General idea of finding an approximation  $u(x)$  to some given  $f(x)$ :

$$u(x) = \sum_{i=0}^N c_i \psi_i(x)$$

where

- $\psi_i(x)$  are prescribed functions
- $c_i$ ,  $i = 0, \dots, N$  are unknown coefficients to be determined



# How to determine the coefficients?

We shall address three approaches:

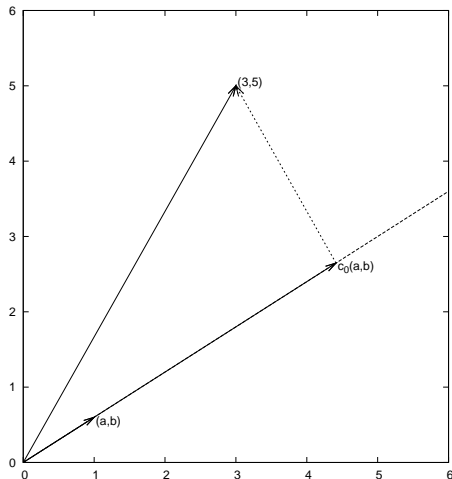
- The least squares method
- The projection (or Galerkin) method
- The interpolation (or collocation) method

Underlying motivation for our notation.

Our mathematical framework for doing this is phrased in a way such that it becomes easy to understand and use the [FEniCS](#) software package for finite element computing.

# Approximation of planar vectors; problem

Given a vector  $\mathbf{f} = (3, 5)$ , find an approximation to  $\mathbf{f}$  directed along a given line.



# Approximation of planar vectors; vector space terminology

$$V = \text{span} \{ \psi_0 \}$$

- $\psi_0$  is a basis vector in the space  $V$
- Seek  $\mathbf{u} = c_0 \psi_0 \in V$
- Determine  $c_0$  such that  $\mathbf{u}$  is the "best" approximation to  $\mathbf{f}$
- Visually, "best" is obvious

Define

- the error  $\mathbf{e} = \mathbf{f} - \mathbf{u}$
- the (Euclidean) scalar product of two vectors:  $(\mathbf{u}, \mathbf{v})$
- the norm of  $\mathbf{e}$ :  $\|\mathbf{e}\| = \sqrt{(\mathbf{e}, \mathbf{e})}$

# The least squares method; principle

- Idea: find  $c_0$  such that  $||\mathbf{e}||$  is minimized
- Mathematical convenience: minimize  $E = ||\mathbf{e}||^2$

$$\frac{\partial E}{\partial c_0} = 0$$

# The least squares method; calculations

$$\begin{aligned} E(c_0) &= (\mathbf{e}, \mathbf{e}) = (\mathbf{f} - \mathbf{u}, \mathbf{f} - \mathbf{u}) = (\mathbf{f} - c_0\psi_0, \mathbf{f} - c_0\psi_0) \\ &= (\mathbf{f}, \mathbf{f}) - 2c_0(\mathbf{f}, \psi_0) + c_0^2(\psi_0, \psi_0) \end{aligned}$$

$$\frac{\partial E}{\partial c_0} = -2(\mathbf{f}, \psi_0) + 2c_0(\psi_0, \psi_0) = 0 \quad (1)$$

$$c_0 = \frac{(\mathbf{f}, \psi_0)}{(\psi_0, \psi_0)} = \frac{3a + 5b}{a^2 + b^2}$$

Observation to be used later: the vanishing derivative (1) can be alternatively written as

$$(\mathbf{e}, \psi_0) = 0$$

# The projection (or Galerkin) method

- Last slide:  $\min E$  is equivalent with  $(\mathbf{e}, \psi_0) = 0$
- $(\mathbf{e}, \psi_0) = 0$  implies  $(\mathbf{e}, \mathbf{v}) = 0$  for *any*  $\mathbf{v} \in V$
- That is: instead of using the least-squares principle, we can require that  $\mathbf{e}$  is orthogonal to *any*  $\mathbf{v} \in V$   
(visually clear, but can easily be computed too)
- Precise math: find  $c_0$  such that  $(\mathbf{e}, \mathbf{v}) = 0, \quad \forall \mathbf{v} \in V$
- Equivalent (see notes): find  $c_0$  such that  $(\mathbf{e}, \psi_0) = 0$
- Insert  $\mathbf{e} = \mathbf{f} - c_0 \psi_0$  and solve for  $c_0$
- Same equation for  $c_0$  and hence same solution as in the least squares method

# The projection (or Galerkin) method

- Last slide:  $\min E$  is equivalent with  $(\mathbf{e}, \psi_0) = 0$
- $(\mathbf{e}, \psi_0) = 0$  implies  $(\mathbf{e}, \mathbf{v}) = 0$  for *any*  $\mathbf{v} \in V$
- That is: instead of using the least-squares principle, we can require that  $\mathbf{e}$  is orthogonal to *any*  $\mathbf{v} \in V$   
(visually clear, but can easily be computed too)
- Precise math: find  $c_0$  such that  $(\mathbf{e}, \mathbf{v}) = 0, \quad \forall \mathbf{v} \in V$
- Equivalent (see notes): find  $c_0$  such that  $(\mathbf{e}, \psi_0) = 0$
- Insert  $\mathbf{e} = \mathbf{f} - c_0 \psi_0$  and solve for  $c_0$
- Same equation for  $c_0$  and hence same solution as in the least squares method

# The projection (or Galerkin) method

- Last slide:  $\min E$  is equivalent with  $(\mathbf{e}, \psi_0) = 0$
- $(\mathbf{e}, \psi_0) = 0$  implies  $(\mathbf{e}, \mathbf{v}) = 0$  for *any*  $\mathbf{v} \in V$
- That is: instead of using the least-squares principle, we can require that  $\mathbf{e}$  is orthogonal to *any*  $\mathbf{v} \in V$   
(visually clear, but can easily be computed too)
- Precise math: find  $c_0$  such that  $(\mathbf{e}, \mathbf{v}) = 0, \quad \forall \mathbf{v} \in V$
- Equivalent (see notes): find  $c_0$  such that  $(\mathbf{e}, \psi_0) = 0$
- Insert  $\mathbf{e} = \mathbf{f} - c_0 \psi_0$  and solve for  $c_0$
- Same equation for  $c_0$  and hence same solution as in the least squares method



# The projection (or Galerkin) method

- Last slide:  $\min E$  is equivalent with  $(\mathbf{e}, \psi_0) = 0$
- $(\mathbf{e}, \psi_0) = 0$  implies  $(\mathbf{e}, \mathbf{v}) = 0$  for *any*  $\mathbf{v} \in V$
- That is: instead of using the least-squares principle, we can require that  $\mathbf{e}$  is orthogonal to *any*  $\mathbf{v} \in V$   
(visually clear, but can easily be computed too)
- Precise math: find  $c_0$  such that  $(\mathbf{e}, \mathbf{v}) = 0, \quad \forall \mathbf{v} \in V$
- Equivalent (see notes): find  $c_0$  such that  $(\mathbf{e}, \psi_0) = 0$
- Insert  $\mathbf{e} = \mathbf{f} - c_0 \psi_0$  and solve for  $c_0$
- Same equation for  $c_0$  and hence same solution as in the least squares method

# The projection (or Galerkin) method

- Last slide:  $\min E$  is equivalent with  $(\mathbf{e}, \psi_0) = 0$
- $(\mathbf{e}, \psi_0) = 0$  implies  $(\mathbf{e}, \mathbf{v}) = 0$  for *any*  $\mathbf{v} \in V$
- That is: instead of using the least-squares principle, we can require that  $\mathbf{e}$  is orthogonal to *any*  $\mathbf{v} \in V$   
(visually clear, but can easily be computed too)
- Precise math: find  $c_0$  such that  $(\mathbf{e}, \mathbf{v}) = 0, \quad \forall \mathbf{v} \in V$
- Equivalent (see notes): find  $c_0$  such that  $(\mathbf{e}, \psi_0) = 0$
- Insert  $\mathbf{e} = \mathbf{f} - c_0 \psi_0$  and solve for  $c_0$
- Same equation for  $c_0$  and hence same solution as in the least squares method

# The projection (or Galerkin) method

- Last slide:  $\min E$  is equivalent with  $(\mathbf{e}, \psi_0) = 0$
- $(\mathbf{e}, \psi_0) = 0$  implies  $(\mathbf{e}, \mathbf{v}) = 0$  for *any*  $\mathbf{v} \in V$
- That is: instead of using the least-squares principle, we can require that  $\mathbf{e}$  is orthogonal to *any*  $\mathbf{v} \in V$   
(visually clear, but can easily be computed too)
- Precise math: find  $c_0$  such that  $(\mathbf{e}, \mathbf{v}) = 0, \quad \forall \mathbf{v} \in V$
- Equivalent (see notes): find  $c_0$  such that  $(\mathbf{e}, \psi_0) = 0$
- Insert  $\mathbf{e} = \mathbf{f} - c_0 \psi_0$  and solve for  $c_0$
- Same equation for  $c_0$  and hence same solution as in the least squares method

# The projection (or Galerkin) method

- Last slide:  $\min E$  is equivalent with  $(\mathbf{e}, \psi_0) = 0$
- $(\mathbf{e}, \psi_0) = 0$  implies  $(\mathbf{e}, \mathbf{v}) = 0$  for *any*  $\mathbf{v} \in V$
- That is: instead of using the least-squares principle, we can require that  $\mathbf{e}$  is orthogonal to *any*  $\mathbf{v} \in V$   
(visually clear, but can easily be computed too)
- Precise math: find  $c_0$  such that  $(\mathbf{e}, \mathbf{v}) = 0, \quad \forall \mathbf{v} \in V$
- Equivalent (see notes): find  $c_0$  such that  $(\mathbf{e}, \psi_0) = 0$
- Insert  $\mathbf{e} = \mathbf{f} - c_0 \psi_0$  and solve for  $c_0$
- Same equation for  $c_0$  and hence same solution as in the least squares method

# The projection (or Galerkin) method

- Last slide:  $\min E$  is equivalent with  $(\mathbf{e}, \psi_0) = 0$
- $(\mathbf{e}, \psi_0) = 0$  implies  $(\mathbf{e}, \mathbf{v}) = 0$  for *any*  $\mathbf{v} \in V$
- That is: instead of using the least-squares principle, we can require that  $\mathbf{e}$  is orthogonal to *any*  $\mathbf{v} \in V$   
(visually clear, but can easily be computed too)
- Precise math: find  $c_0$  such that  $(\mathbf{e}, \mathbf{v}) = 0, \quad \forall \mathbf{v} \in V$
- Equivalent (see notes): find  $c_0$  such that  $(\mathbf{e}, \psi_0) = 0$
- Insert  $\mathbf{e} = \mathbf{f} - c_0 \psi_0$  and solve for  $c_0$
- Same equation for  $c_0$  and hence same solution as in the least squares method

# Approximation of general vectors

Given a vector  $\mathbf{f}$ , find an approximation  $\mathbf{u} \in V$ :

$$V = \text{span} \{ \psi_0, \dots, \psi_N \}$$

We have a set of linearly independent basis vectors  $\psi_0, \dots, \psi_N$ .  
Any  $\mathbf{u} \in V$  can then be written as

$$\mathbf{u} = \sum_{j=0}^N c_j \psi_j$$

# The least squares method

Idea: find  $c_0, \dots, c_N$  such that  $E = \|\mathbf{e}\|^2$  is minimized,  $\mathbf{e} = \mathbf{f} - \mathbf{u}$ .

$$\begin{aligned} E(c_0, \dots, c_N) &= (\mathbf{e}, \mathbf{e}) = \left( \mathbf{f} - \sum_j c_j \psi_j, \mathbf{f} - \sum_j c_j \psi_j \right) \\ &= (\mathbf{f}, \mathbf{f}) - 2 \sum_{j=0}^N c_j (\mathbf{f}, \psi_j) + \sum_{p=0}^N \sum_{q=0}^N c_p c_q (\psi_p, \psi_q) \end{aligned}$$

$$\frac{\partial E}{\partial c_i} = 0, \quad i = 0, \dots, N$$

After some work we end up with a *linear system*

$$\sum_{j=0}^N A_{i,j} c_j = b_i, \quad i = 0, \dots, N \quad (2)$$

$$A_{i,j} = (\psi_i, \psi_j) \quad (3)$$

$$b_i = (\mathbf{f}, \psi_i) \quad (4)$$

# The projection (or Galerkin) method

Can be shown that minimizing  $\|\mathbf{e}\|$  implies that  $\mathbf{e}$  is orthogonal to all  $\mathbf{v} \in V$ :

$$(\mathbf{e}, \mathbf{v}) = 0, \quad \forall \mathbf{v} \in V$$

which implies that  $\mathbf{e}$  must be orthogonal to each basis vector:

$$(\mathbf{e}, \psi_i) = 0, \quad i = 0, \dots, N$$

This orthogonality condition is the principle of the projection (or Galerkin) method. Leads to the same linear system as in the least squares method.



# Approximation of functions

Let  $V$  be a *function space* spanned by a set of *basis functions*  $\psi_0, \dots, \psi_N$ ,

$$V = \text{span} \{ \psi_0, \dots, \psi_N \}$$

Find  $u \in V$  as a linear combination of the basis functions:

$$u = \sum_{j \in \mathcal{I}_s} c_j \psi_j, \quad \mathcal{I}_s = \{0, 1, \dots, N\}$$

# The least squares method; extend the basic ideas to functions

Extend the ideas from the vector case: minimize the (square) norm of the error,  $E$ , with respect to the coefficients  $c_j$ ,  $j \in \mathcal{I}_s$ :

$$E = (e, e) = (f - u, f - u) = (f(x) - \sum_{j \in \mathcal{I}_s} c_j \psi_j(x), f(x) - \sum_{j \in \mathcal{I}_s} c_j \psi_j(x))$$

$$\frac{\partial E}{\partial c_i} = 0, \quad i \in \mathcal{I}_s$$

What scalar product?

$$(f, g) = \int_{\Omega} f(x)g(x) dx$$

(natural extension from Euclidean product  $(\mathbf{u}, \mathbf{v}) = \sum_j u_j v_j$ )

# The least squares method; details

$$\begin{aligned} E(c_0, \dots, c_N) &= (e, e) = (f - u, f - u) \\ &= (f, f) - 2 \sum_{j \in \mathcal{I}_s} c_j (f, \psi_j) + \sum_{p \in \mathcal{I}_s} \sum_{q \in \mathcal{I}_s} c_p c_q (\psi_p, \psi_q) \end{aligned}$$

$$\frac{\partial E}{\partial c_i} = 0, \quad i \in \mathcal{I}_s$$

The computations are *identical to the vector case*, and consequently we get a linear system

$$\sum_{j \in \mathcal{I}_s}^N A_{i,j} c_j = b_i, \quad i \in \mathcal{I}_s, \quad A_{i,j} = (\psi_i, \psi_j), \quad b_i = (f, \psi_i)$$

# The projection (or Galerkin) method

As before, minimizing  $(e, e)$  is equivalent to

$$(e, \psi_i) = 0, \quad i \in \mathcal{I}_s$$

which is equivalent to

$$(e, v) = 0, \quad \forall v \in V$$

which is the projection (or Galerkin) method.

The algebra is the same as in the multi-dimensional vector case, and we get the same linear system as arose from the least squares method.

## Example: linear approximation; problem

Problem.

Approximate a parabola  $f(x) = 10(x - 1)^2 - 1$  by a straight line.

$$V = \text{span} \{1, x\}$$

That is,  $\psi_0(x) = 1$ ,  $\psi_1(x) = x$ , and  $N = 1$ . We seek

$$u = c_0\psi_0(x) + c_1\psi_1(x) = c_0 + c_1x$$

## Example: linear approximation; solution

$$A_{0,0} = (\psi_0, \psi_0) = \int_1^2 1 \cdot 1 \, dx = 1$$

$$A_{0,1} = (\psi_0, \psi_1) = \int_1^2 1 \cdot x \, dx = 3/2$$

$$A_{1,0} = A_{0,1} = 3/2$$

$$A_{1,1} = (\psi_1, \psi_1) = \int_1^2 x \cdot x \, dx = 7/3$$

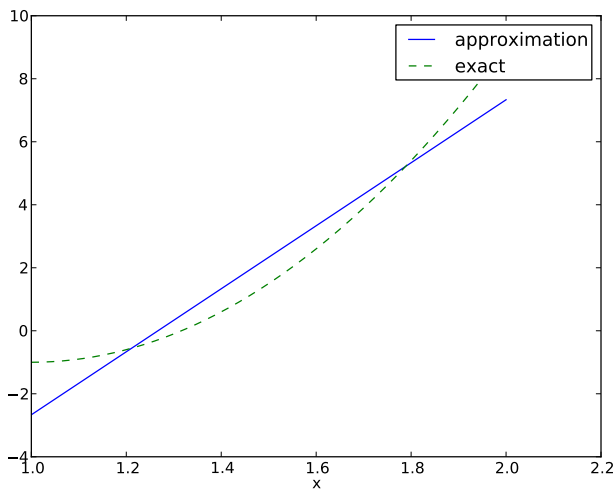
$$b_1 = (f, \psi_0) = \int_1^2 (10(x-1)^2 - 1) \cdot 1 \, dx = 7/3$$

$$b_2 = (f, \psi_1) = \int_1^2 (10(x-1)^2 - 1) \cdot x \, dx = 13/3$$

Solution of 2x2 linear system:

$$c_0 = -38/3, \quad c_1 = 10, \quad u(x) = 10x - \frac{38}{3}$$

## Example: linear approximation; plot



# Implementation of the least squares method; ideas

Consider symbolic computation of the linear system, where

- $f(x)$  is given as a sympy expression  $f$  (involving the symbol  $x$ ),
- $\text{psi}$  is a list of  $\{\psi_i\}_{i \in \mathcal{I}_s}$ ,
- $\Omega$  is a 2-tuple/list holding the domain  $\Omega$

Carry out the integrations, solve the linear system, and return

$$u(x) = \sum_j c_j \psi_j(x)$$



# Implementation of the least squares method; symbolic code

```
import sympy as sp

def least_squares(f, psi, Omega):
    N = len(psi) - 1
    A = sp.zeros((N+1, N+1))
    b = sp.zeros((N+1, 1))
    x = sp.Symbol('x')
    for i in range(N+1):
        for j in range(i, N+1):
            A[i,j] = sp.integrate(psi[i]*psi[j],
                                   (x, Omega[0], Omega[1]))
            A[j,i] = A[i,j]
        b[i,0] = sp.integrate(psi[i]*f, (x, Omega[0], Omega[1]))
    c = A.LUsolve(b)
    u = 0
    for i in range(len(psi)):
        u += c[i,0]*psi[i]
    return u, c
```

Observe: symmetric coefficient matrix so we can halve the integrations.

## Alternative: numerical code

- Symbolic integration may be impossible and/or very slow
- Turn to pure numerical computations in those cases
- Supply Python functions  $f(x)$ ,  $\psi(x,i)$ , and a mesh  $x$

```
def least_squares_numerical(f, psi, N, x,
                           integration_method='scipy',
                           orthogonal_basis=False):

    import scipy.integrate
    A = np.zeros((N+1, N+1))
    b = np.zeros(N+1)
    Omega = [x[0], x[-1]]
    dx = x[1] - x[0]

    for i in range(N+1):
        j_limit = i+1 if orthogonal_basis else N+1
        for j in range(i, j_limit):
            print '(%d,%d)' % (i, j)
            if integration_method == 'scipy':
                A_ij = scipy.integrate.quad(
                    lambda x: psi(x,i)*psi(x,j),
                    Omega[0], Omega[1], epsabs=1E-9, epsrel=1E-9)[0]
            elif ...
            A[i,j] = A[j,i] = A_ij
```

## Numerical code continued

```
def least_squares_numerical(f, psi, N, x,
                           integration_method='scipy',
                           orthogonal_basis=False):
    ...
    for i in range(N+1):
        ...
        if integration_method == 'scipy':
            b_i = scipy.integrate.quad(
                lambda x: f(x)*psi(x,i), Omega[0], Omega[1],
                epsabs=1E-9, epsrel=1E-9)[0]
        elif ...
        b[i] = b_i

    c = b/np.diag(A) if orthogonal_basis else np.linalg.solve(A, b)
    u = sum(c[i]*psi(x, i) for i in range(N+1))
    return u, c
```

# Implementation of the least squares method; plotting

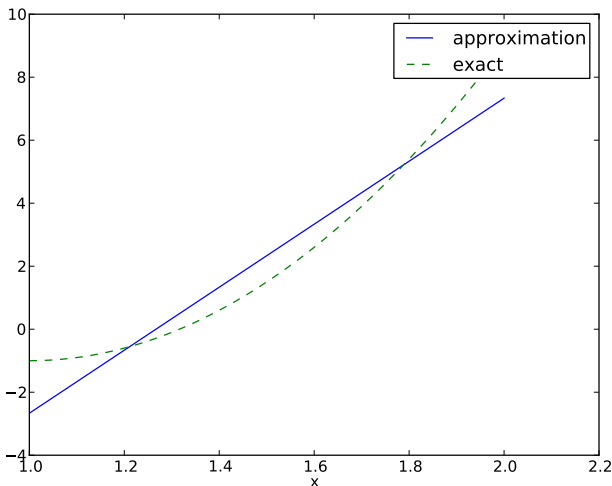
Compare  $f$  and  $u$  visually:

```
def comparison_plot(f, u, Omega, filename='tmp.pdf'):
    x = sp.Symbol('x')
    # Turn f and u to ordinary Python functions
    f = sp.lambdify([x], f, modules="numpy")
    u = sp.lambdify([x], u, modules="numpy")
    resolution = 401 # no of points in plot
    xcoor = linspace(Omega[0], Omega[1], resolution)
    exact = f(xcoor)
    approx = u(xcoor)
    plot(xcoor, approx)
    hold('on')
    plot(xcoor, exact)
    legend(['approximation', 'exact'])
    savefig(filename)
```

All code in module `approx1D.py`

# Implementation of the least squares method; application

```
>>> from approx1D import *  
>>> x = sp.Symbol('x')  
>>> f = 10*(x-1)**2-1  
>>> u, c = least_squares(f=f, psi=[1, x], Omega=[1, 2])  
>>> comparison_plot(f, u, Omega=[1, 2])
```



# Perfect approximation; parabola approximating parabola

- What if we add  $\psi_2 = x^2$  to the space  $V$ ?
- That is, approximating a parabola by any parabola?
- (Hopefully we get the exact parabola!)

```
>>> from approx1D import *
>>> x = sp.Symbol('x')
>>> f = 10*(x-1)**2-1
>>> u, c = least_squares(f=f, psi=[1, x, x**2], Omega=[1, 2])
>>> print u
10*x**2 - 20*x + 9
>>> print sp.expand(f)
10*x**2 - 20*x + 9
```

## Perfect approximation; the general result

- What if we use  $\psi_i(x) = x^i$  for  $i = 0, \dots, N = 40$ ?
- The output from `least_squares` is  $c_i = 0$  for  $i > 2$

### General result.

If  $f \in V$ , least squares and projection/Galerkin give  $u = f$ .

## Perfect approximation; proof of the general result

If  $f \in V$ ,  $f = \sum_{j \in \mathcal{I}_s} d_j \psi_j$ , for some  $\{d_i\}_{i \in \mathcal{I}_s}$ . Then

$$b_i = (f, \psi_i) = \sum_{j \in \mathcal{I}_s} d_j (\psi_j, \psi_i) = \sum_{j \in \mathcal{I}_s} d_j A_{i,j}$$

The linear system  $\sum_j A_{i,j} c_j = b_i$ ,  $i \in \mathcal{I}_s$ , is then

$$\sum_{j \in \mathcal{I}_s} c_j A_{i,j} = \sum_{j \in \mathcal{I}_s} d_j A_{i,j}, \quad i \in \mathcal{I}_s$$

which implies that  $c_i = d_i$  for  $i \in \mathcal{I}_s$  and  $u$  is identical to  $f$ .



The previous computations were symbolic. What if we solve the linear system numerically with standard arrays?

That is,  $f$  is parabola, but we approximate with

$$u(x) = c_0 + c_1x + c_2x^2 + c_3x^3 + \cdots + c_Nx^N$$

We expect  $c_2 = c_3 = \cdots = c_N = 0$  since  $f \in V$  implies  $u = f$ .

Will we get this result with finite precision computer arithmetic?

# Finite-precision/numerical computations; results

exact	sympy	numpy32	numpy64
9	9.62	5.57	8.98
-20	-23.39	-7.65	-19.93
10	17.74	-4.50	9.96
0	-9.19	4.13	-0.26
0	5.25	2.99	0.72
0	0.18	-1.21	-0.93
0	-2.48	-0.41	0.73
0	1.81	-0.013	-0.36
0	-0.66	0.08	0.11
0	0.12	0.04	-0.02
0	-0.001	-0.02	0.002

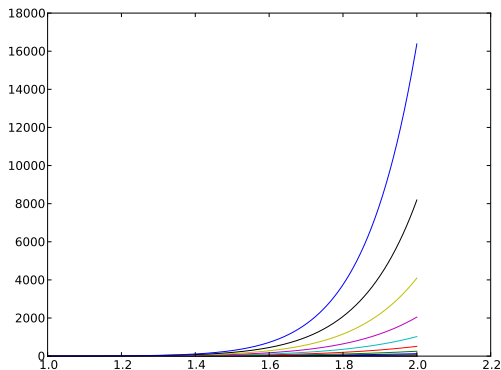
- Column 2: `matrix` and `lu_solve` from `sympy.mpmath.fp`
- Column 3: numpy matrix with 4-byte floats
- Column 4: numpy matrix with 8-byte floats

### Ill-conditioning: observation in this special case

Observations:

- Significant round-off errors in the numerical computations (!)
- But if we plot the approximations they look good (!)

The source of the problem is that the basis functions  $x^i$  become almost linearly dependent for large  $N$ .



## Ill-conditioning: general conclusions

- Almost linearly dependent basis functions give almost singular matrices
- Such matrices are said to be *ill conditioned*, and Gaussian elimination is severely affected by round-off errors
- The basis  $1, x, x^2, x^3, x^4, \dots$  is a bad basis
- Polynomials are fine as basis, but the more orthogonal they are,  $(\psi_i, \psi_j) \approx 0$ , the better

# Fourier series approximation; problem and code

Let's approximate  $f$  by a typical Fourier series expansion

$$u(x) = \sum_i a_i \sin i\pi x = \sum_{j=0}^N c_j \sin((j+1)\pi x)$$

which means that

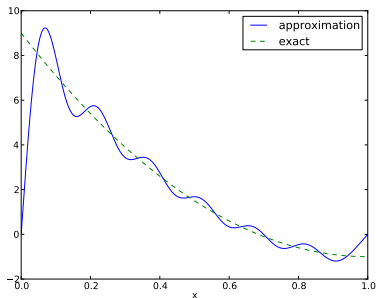
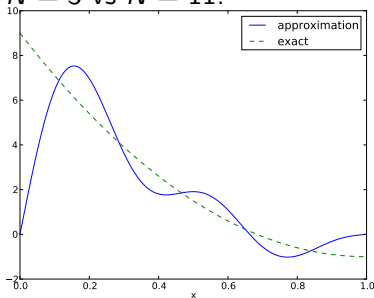
$$V = \text{span} \{ \sin \pi x, \sin 2\pi x, \dots, \sin(N+1)\pi x \}$$

Computations using the `least_squares` function:

```
N = 3
from sympy import sin, pi
psi = [sin(pi*(i+1)*x) for i in range(N+1)]
f = 10*(x-1)**2 - 1
Omega = [0, 1]
u, c = least_squares(f, psi, Omega)
comparison_plot(f, u, Omega)
```

# Fourier series approximation; plot

$N = 3$  vs  $N = 11$ :



**Problem:**

All  $\psi_i(0) = 0$  and hence  $u(0) = 0 \neq f(0) = 9$ . Similar problem at  $x = 1$ . The boundary values of  $u$  are always wrong!

# Fourier series approximation; improvements

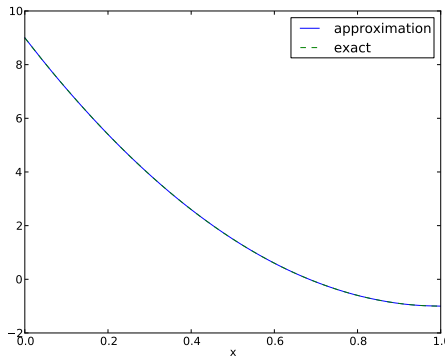
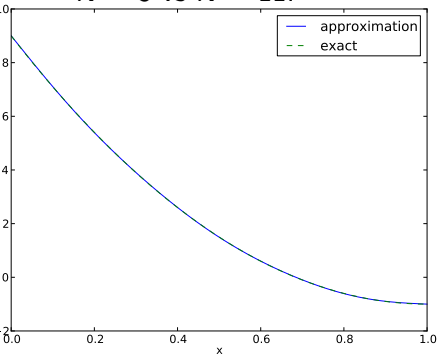
- Considerably improvement with  $N = 11$ , but still undesired discrepancy at  $x = 0$  and  $x = 1$
- Possible remedy: add a term that leads to correct boundary values

$$u(x) = f(0)(1 - x) + xf(1) + \sum_{j \in \mathcal{I}_s} c_j \psi_j(x)$$

The extra terms ensure  $u(0) = f(0)$  and  $u(1) = f(1)$  and is a strikingly good help to get a good approximation!

# Fourier series approximation; final results

$N = 3$  vs  $N = 11$ :





# Orthogonal basis functions

This choice of sine functions as basis functions is popular because

- the basis functions are orthogonal:  $(\psi_i, \psi_j) = 0$
- implying that  $A_{i,j}$  is a diagonal matrix
- implying that we can solve for  $c_i = 2 \int_0^1 f(x) \sin((i+1)\pi x) dx$
- and what we get is the standard Fourier sine series of  $f$

*In general*, for an orthogonal basis,  $A_{i,j}$  is diagonal and we can easily solve for  $c_i$ :

$$c_i = \frac{b_i}{A_{i,i}} = \frac{(f, \psi_i)}{(\psi_i, \psi_i)}$$

# The collocation or interpolation method; ideas and math

Here is another idea for approximating  $f(x)$  by  $u(x) = \sum_j c_j \psi_j$ :

- Force  $u(x_i) = f(x_i)$  at some selected *collocation* points  $\{x_i\}_{i \in \mathcal{I}_s}$
- Then  $u$  is said to *interpolate*  $f$
- The method is known as *interpolation* or *collocation*

$$u(x_i) = \sum_{j \in \mathcal{I}_s} c_j \psi_j(x_i) = f(x_i) \quad i \in \mathcal{I}_s, N$$

This is a linear system with no need for integration:

$$\sum_{j \in \mathcal{I}_s} A_{i,j} c_j = b_i, \quad i \in \mathcal{I}_s \tag{5}$$

$$A_{i,j} = \psi_j(x_i) \tag{6}$$

$$b_i = f(x_i) \tag{7}$$

No symmetric matrix:  $\psi_j(x_i) \neq \psi_i(x_j)$  in general

# The collocation or interpolation method; implementation

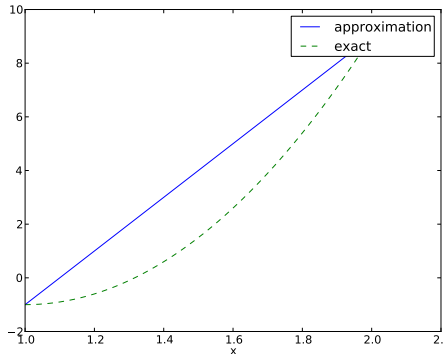
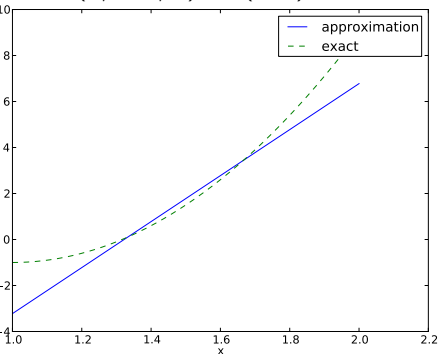
points holds the interpolation/collocation points

```
def interpolation(f, psi, points):
    N = len(psi) - 1
    A = sp.zeros((N+1, N+1))
    b = sp.zeros((N+1, 1))
    x = sp.Symbol('x')
    # Turn psi and f into Python functions
    psi = [sp.lambdify([x], psi[i]) for i in range(N+1)]
    f = sp.lambdify([x], f)
    for i in range(N+1):
        for j in range(N+1):
            A[i,j] = psi[j](points[i])
        b[i,0] = f(points[i])
    c = A.LUsolve(b)
    u = 0
    for i in range(len(psi)):
        u += c[i,0]*psi[i](x)
    return u
```

# The collocation or interpolation method; approximating a parabola by linear functions

- Potential difficulty: how to choose  $x_i$ ?
- The results are sensitive to the points!

$(4/3, 5/3)$  vs  $(1, 2)$ :



# Lagrange polynomials; motivation and ideas

Motivation:

- The interpolation/collocation method avoids integration
- With a diagonal matrix  $A_{i,j} = \psi_j(x_i)$  we can solve the linear system by hand

The *Lagrange interpolating polynomials*  $\psi_j$  have the property that

$$\psi_i(x_j) = \delta_{ij}, \quad \delta_{ij} = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}$$

Hence,  $c_i = f(x_i)$  and

$$u(x) = \sum_{j \in \mathcal{I}_s} f(x_j) \psi_j(x)$$

- Lagrange polynomials and interpolation/collocation look convenient

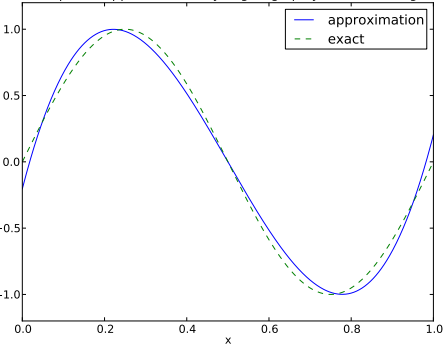
# Lagrange polynomials; formula and code

$$\psi_i(x) = \prod_{j=0, j \neq i}^N \frac{x - x_j}{x_i - x_j} = \frac{x - x_0}{x_i - x_0} \dots \frac{x - x_{i-1}}{x_i - x_{i-1}} \frac{x - x_{i+1}}{x_i - x_{i+1}} \dots \frac{x - x_N}{x_i - x_N}$$

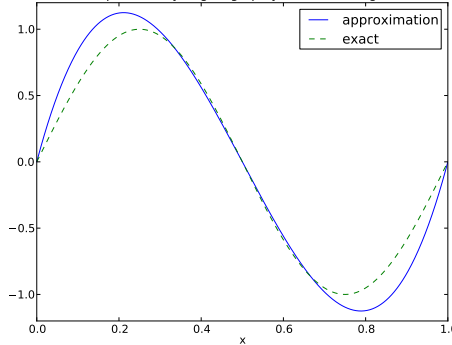
```
def Lagrange_polynomial(x, i, points):  
    p = 1  
    for k in range(len(points)):  
        if k != i:  
            p *= (x - points[k])/(points[i] - points[k])  
    return p
```

# Lagrange polynomials; successful example

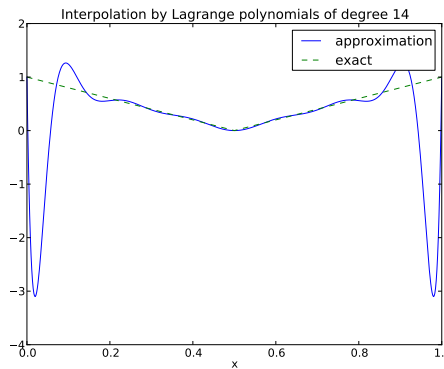
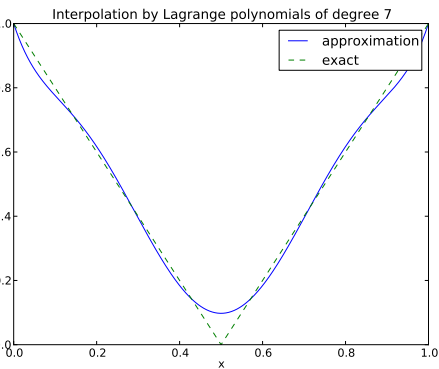
Least squares approximation by Lagrange polynomials of degree 3



Interpolation by Lagrange polynomials of degree 3



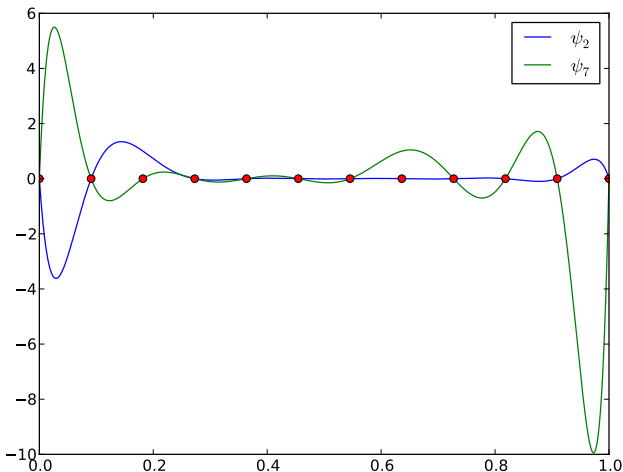
# Lagrange polynomials; a less successful example





# Lagrange polynomials; oscillatory behavior

12 points, degree 11, plot of two of the Lagrange polynomials - note that they are zero at all points except one.



Problem: strong oscillations near the boundaries for larger  $N$

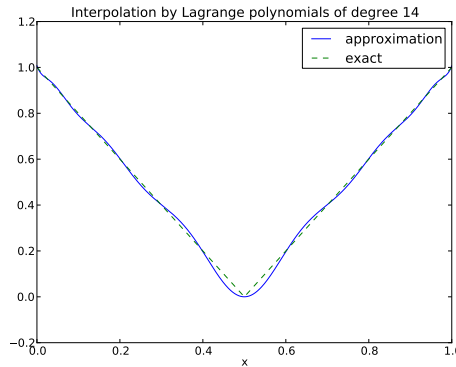
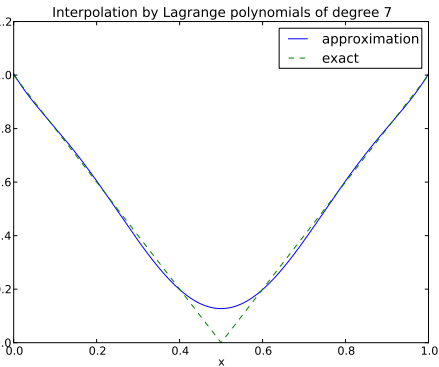
## Lagrange polynomials; remedy for strong oscillations

The oscillations can be reduced by a more clever choice of interpolation points, called the *Chebyshev nodes*:

$$x_i = \frac{1}{2}(a+b) + \frac{1}{2}(b-a) \cos\left(\frac{2i+1}{2(N+1)}\pi\right), \quad i = 0 \dots, N$$

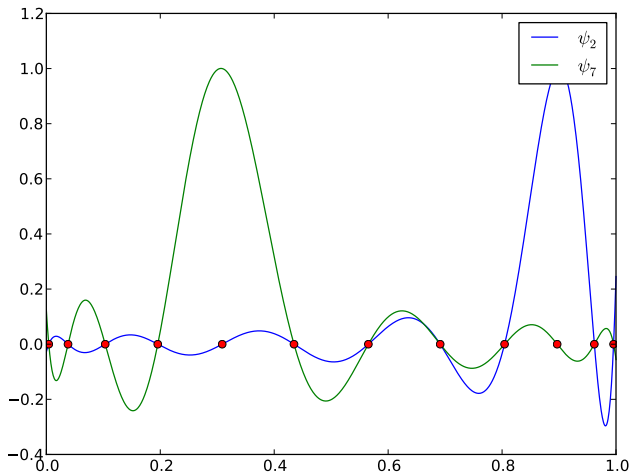
on an interval  $[a, b]$ .

# Lagrange polynomials; recalculation with Chebyshev nodes



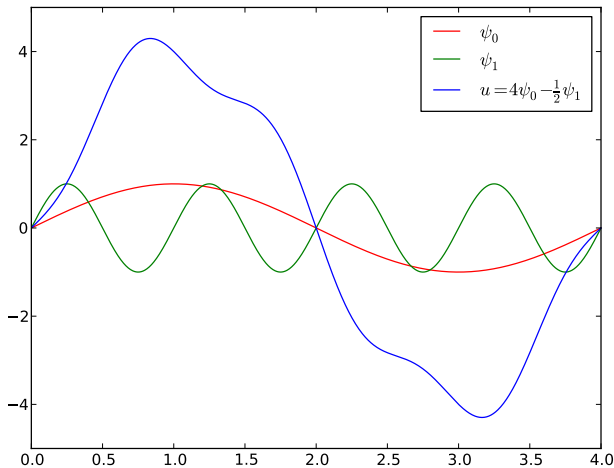
# Lagrange polynomials; less oscillations with Chebyshev nodes

12 points, degree 11, plot of two of the Lagrange polynomials - note that they are zero at all points except one.



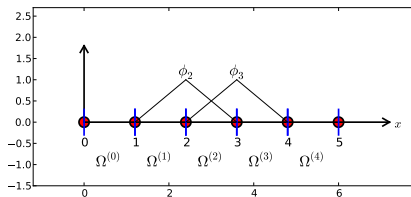
# Finite element basis functions

The basis functions have so far been global:  $\psi_i(x) \neq 0$  almost everywhere

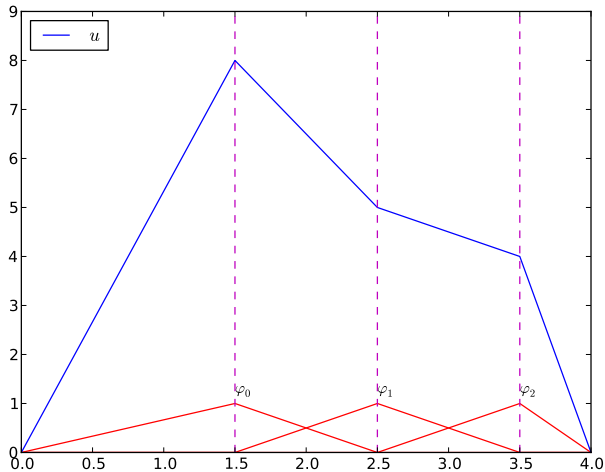


# In the finite element method we use basis functions with local support

- *Local support*:  $\psi_i(x) \neq 0$  for  $x$  in a small subdomain of  $\Omega$
- Typically hat-shaped
- $u(x)$  based on these  $\psi_i$  is a piecewise polynomial defined over many (small) subdomains
- We introduce  $\varphi_i$  as the name of these finite element hat functions (and for now choose  $\psi_i = \varphi_i$ )



The linear combination of hat functions is a piecewise linear function





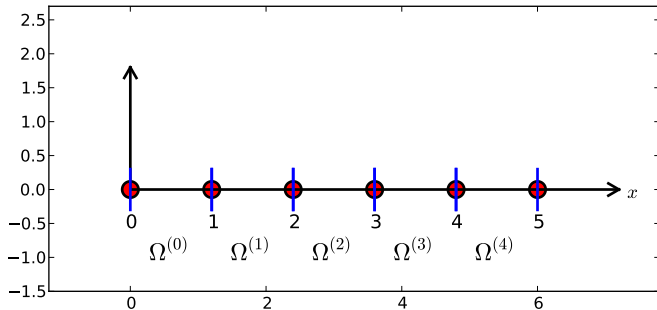
Split  $\Omega$  into non-overlapping subdomains called *elements*:

$$\Omega = \Omega^{(0)} \cup \dots \cup \Omega^{(N_e)}$$

On each element, introduce points called *nodes*:  $x_0, \dots, x_{N_n}$

- The finite element basis functions are named  $\varphi_i(x)$
- $\varphi_i = 1$  at node  $i$  and 0 at all other nodes
- $\varphi_i$  is a Lagrange polynomial on each element
- For nodes at the boundary between two elements,  $\varphi_i$  is made up of a Lagrange polynomial over each element

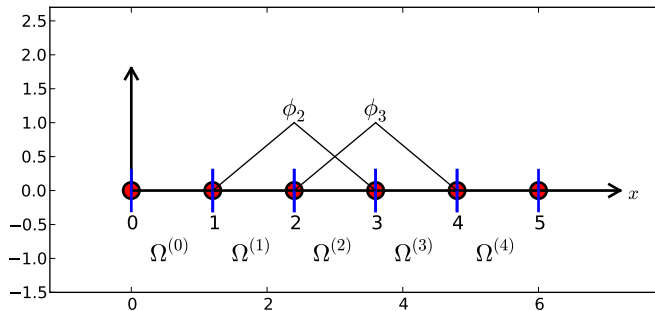
## Example on elements with two nodes (P1 elements)



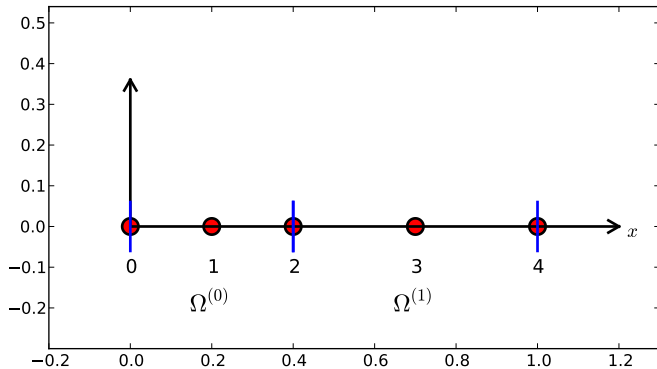
Data structure: `nodes` holds coordinates or nodes, `elements` holds the node numbers in each element

```
nodes = [0, 1.2, 2.4, 3.6, 4.8, 5]  
elements = [[0, 1], [1, 2], [2, 3], [3, 4], [4, 5]]
```

# Illustration of two basis functions on the mesh

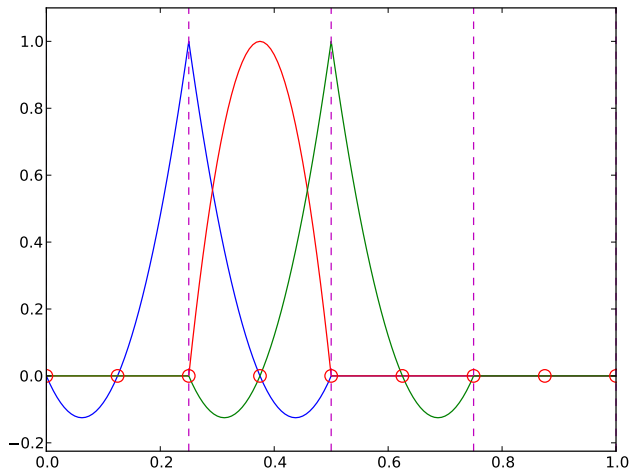


## Example on elements with three nodes (P2 elements)

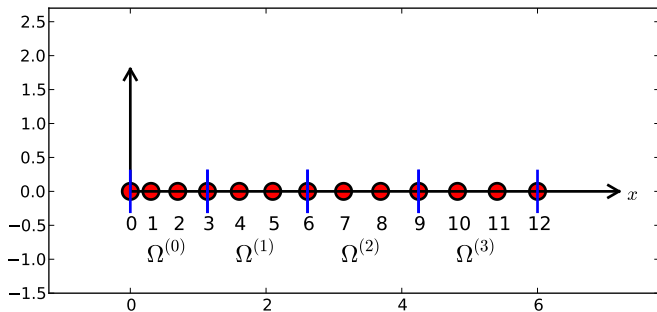


```
nodes = [0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1.0]  
elements = [[0, 1, 2], [2, 3, 4], [4, 5, 6], [6, 7, 8]]
```

# Some corresponding basis functions (P2 elements)

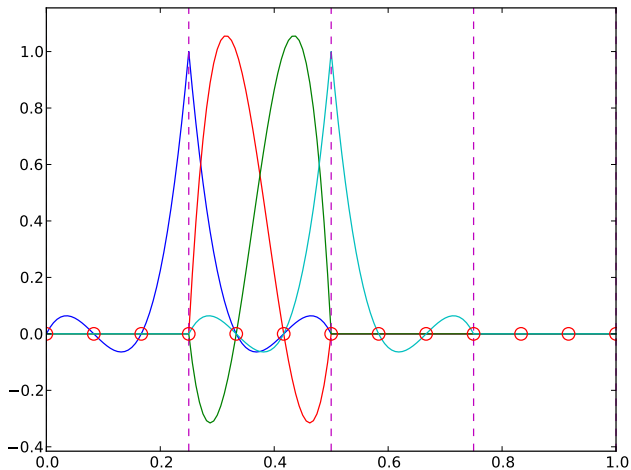


# Examples on elements with four nodes (P3 elements)

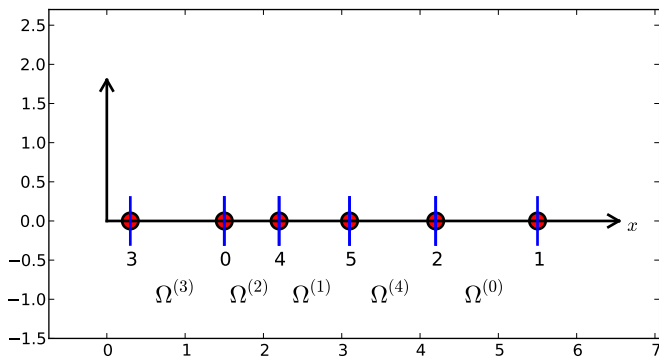


```
d = 3 # d+1 nodes per element
num_elements = 4
num_nodes = num_elements*d + 1
nodes = [i*0.5 for i in range(num_nodes)]
elements = [[i*d+j for j in range(d+1)] for i in range(num_elements)]
```

# Some corresponding basis functions (P3 elements)



The numbering does not need to be regular from left to right



```
nodes = [1.5, 5.5, 4.2, 0.3, 2.2, 3.1]
elements = [[2, 1], [4, 5], [0, 4], [3, 0], [5, 2]]
```



## Interpretation of the coefficients $c_i$

Important property:  $c_i$  is the value of  $u$  at node  $i$ ,  $x_i$ :

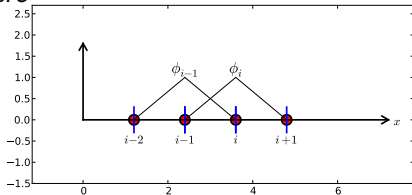
$$u(x_i) = \sum_{j \in \mathcal{I}_s} c_j \varphi_j(x_i) = c_i \varphi_i(x_i) = c_i$$

because  $\varphi_j(x_i) = 0$  if  $i \neq j$

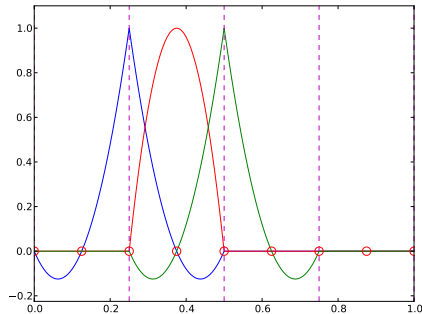
# Properties of the basis functions

- $\varphi_i(x) \neq 0$  only on those elements that contain global node  $i$
- $\varphi_i(x)\varphi_j(x) \neq 0$  if and only if  $i$  and  $j$  are global node numbers in the same element

Since  $A_{i,j} = \int \varphi_i \varphi_j dx$ , *most of the elements in the coefficient matrix will be zero*

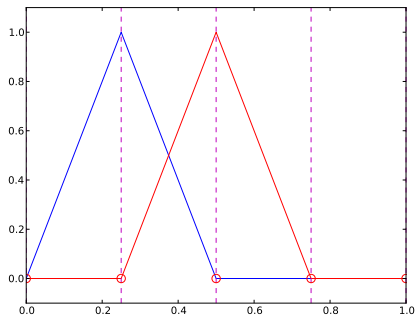


# How to construct quadratic $\varphi_i$ (P2 elements)



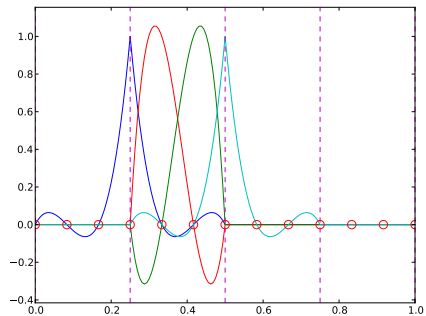
- 1 Associate Lagrange polynomials with the nodes in an element
- 2 When the polynomial is 1 on the element boundary, combine it with the polynomial in the neighboring element

## Example on linear $\varphi_i$ (P1 elements)



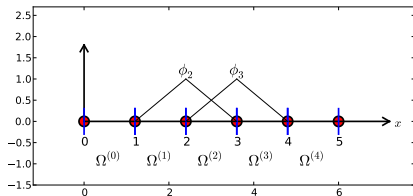
$$\varphi_i(x) = \begin{cases} 0, & x < x_{i-1} \\ (x - x_{i-1})/h, & x_{i-1} \leq x < x_i \\ 1 - (x - x_i)/h, & x_i \leq x < x_{i+1} \\ 0, & x \geq x_{i+1} \end{cases}$$

# Example on cubic $\varphi_i$ (P3 elements)



## Calculating the linear system for $c_i$

# Computing a specific matrix entry (1)

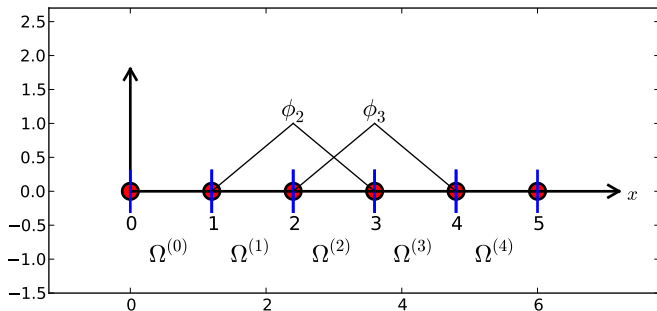


$A_{2,3} = \int_{\Omega} \varphi_2 \varphi_3 dx$ :  $\varphi_2 \varphi_3 \neq 0$  only over element 2. There,

$$\varphi_3(x) = (x - x_2)/h, \quad \varphi_2(x) = 1 - (x - x_2)/h$$

$$A_{2,3} = \int_{\Omega} \varphi_2 \varphi_3 dx = \int_{x_2}^{x_3} \left(1 - \frac{x - x_2}{h}\right) \frac{x - x_2}{h} dx = \frac{h}{6}$$

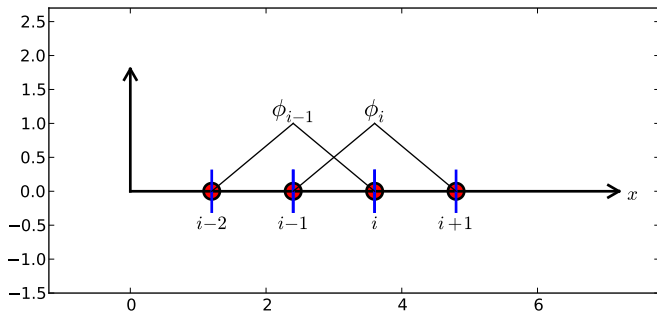
## Computing a specific matrix entry (2)



$$A_{2,2} = \int_{x_1}^{x_2} \left( \frac{x - x_1}{h} \right)^2 dx + \int_{x_2}^{x_3} \left( 1 - \frac{x - x_2}{h} \right)^2 dx = \frac{h}{3}$$



# Calculating a general row in the matrix; figure



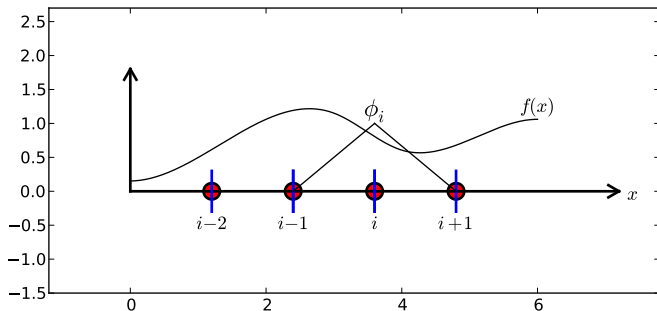
$$A_{i,i-1} = \int_{\Omega} \phi_i \phi_{i-1} dx = ?$$

## Calculating a general row in the matrix; details

$$\begin{aligned} A_{i,i-1} &= \int_{\Omega} \varphi_i \varphi_{i-1} \, dx \\ &= \underbrace{\int_{x_{i-2}}^{x_{i-1}} \varphi_i \varphi_{i-1} \, dx}_{\varphi_i=0} + \int_{x_{i-1}}^{x_i} \varphi_i \varphi_{i-1} \, dx + \underbrace{\int_{x_i}^{x_{i+1}} \varphi_i \varphi_{i-1} \, dx}_{\varphi_{i-1}=0} \\ &= \int_{x_{i-1}}^{x_i} \underbrace{\left( \frac{x - x_i}{h} \right)}_{\varphi_i(x)} \underbrace{\left( 1 - \frac{x - x_{i-1}}{h} \right)}_{\varphi_{i-1}(x)} \, dx = \frac{h}{6} \end{aligned}$$

- $A_{i,i+1} = A_{i,i-1}$  due to symmetry
- $A_{i,i} = h/3$  (same calculation as for  $A_{2,2}$ )
- $A_{0,0} = A_{N,N} = h/3$  (only one element)

# Calculation of the right-hand side



$$b_i = \int_{\Omega} \varphi_i(x) f(x) dx = \int_{x_{i-1}}^{x_i} \frac{x - x_{i-1}}{h} f(x) dx + \int_{x_i}^{x_{i+1}} \left(1 - \frac{x - x_i}{h}\right) f(x) dx$$

Need a specific  $f(x)$  to do more...

## Specific example with two elements; linear system and solution

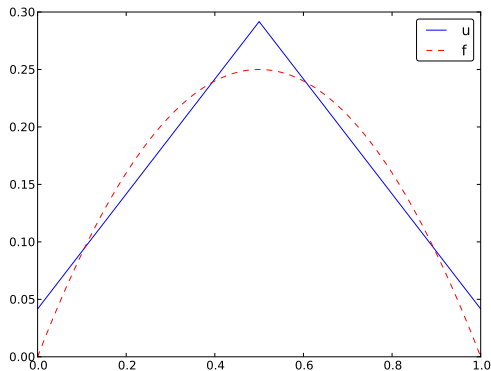
- $f(x) = x(1 - x)$  on  $\Omega = [0, 1]$
- Two equal-sized elements  $[0, 0.5]$  and  $[0.5, 1]$

$$A = \frac{h}{6} \begin{pmatrix} 2 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 2 \end{pmatrix}, \quad b = \frac{h^2}{12} \begin{pmatrix} 2 - 3h \\ 12 - 14h \\ 10 - 17h \end{pmatrix}$$

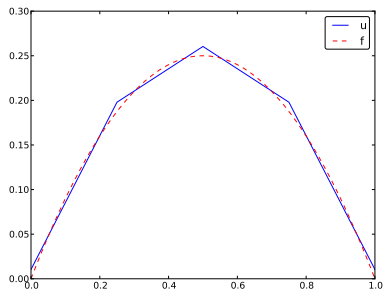
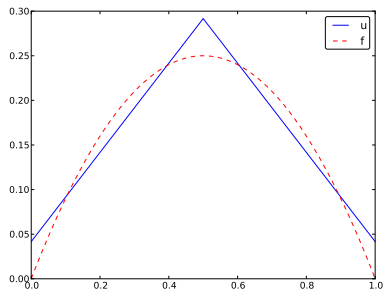
$$c_0 = \frac{h^2}{6}, \quad c_1 = h - \frac{5}{6}h^2, \quad c_2 = 2h - \frac{23}{6}h^2$$

## Specific example with two elements; plot

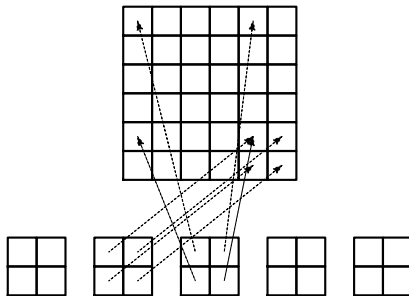
$$u(x) = c_0\varphi_0(x) + c_1\varphi_1(x) + c_2\varphi_2(x)$$



# Specific example: what about four elements?



# Assembly of elementwise computations



# Split the integrals into elementwise integrals

$$A_{i,j} = \int_{\Omega} \varphi_i \varphi_j dx = \sum_e \int_{\Omega^{(e)}} \varphi_i \varphi_j dx, \quad A_{i,j}^{(e)} = \int_{\Omega^{(e)}} \varphi_i \varphi_j dx$$

Important:

- $A_{i,j}^{(e)} \neq 0$  if and only if  $i$  and  $j$  are nodes in element  $e$  (otherwise no overlap between the basis functions)
- all the nonzero elements in  $A_{i,j}^{(e)}$  are collected in an *element matrix*



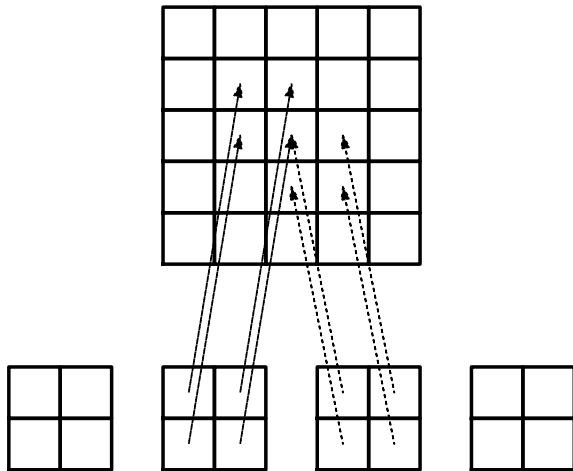
# The element matrix

$$\tilde{A}^{(e)} = \{\tilde{A}_{r,s}^{(e)}\}, \quad \tilde{A}_{r,s}^{(e)} = \int_{\Omega^{(e)}} \varphi_{q(e,r)} \varphi_{q(e,s)} dx, \quad r, s \in I_d = \{0, \dots, d\}$$

- $r, s$  run over *local node numbers* in an element;  $i, j$  run over *global node numbers*
- $i = q(e, r)$ : mapping of local node number  $r$  in element  $e$  to the global node number  $i$  (math equivalent to `i=elements[e][r]`)
- Add  $\tilde{A}_{r,s}^{(e)}$  into the global  $A_{i,j}$  (*assembly*)

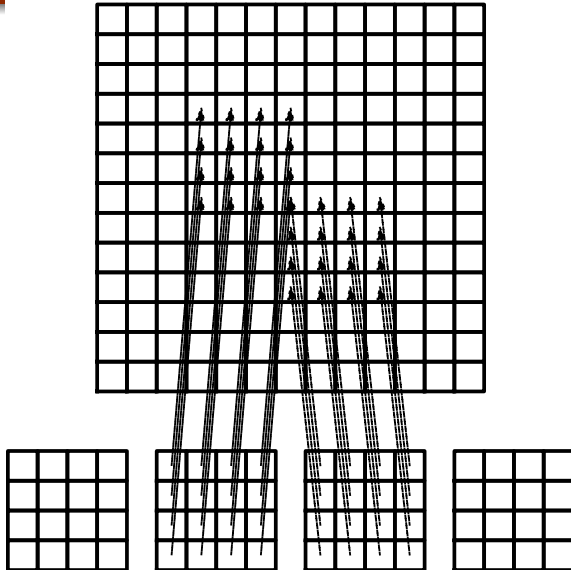
$$A_{q(e,r),q(e,s)} := A_{q(e,r),q(e,s)} + \tilde{A}_{r,s}^{(e)}, \quad r, s \in I_d$$

# Illustration of the matrix assembly: regularly numbered P1 elements

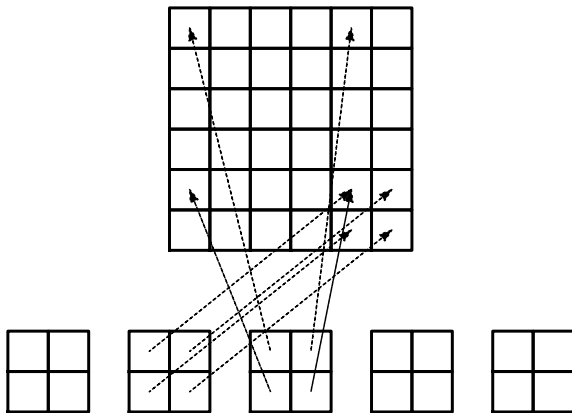


Animation

# Illustration of the matrix assembly: regularly numbered P3 elements



# Illustration of the matrix assembly: irregularly numbered P1 elements



Animation

# Assembly of the right-hand side

$$b_i = \int_{\Omega} f(x) \varphi_i(x) dx = \sum_e \int_{\Omega^{(e)}} f(x) \varphi_i(x) dx, \quad b_i^{(e)} = \int_{\Omega^{(e)}} f(x) \varphi_i(x) dx$$

Important:

- $b_i^{(e)} \neq 0$  if and only if global node  $i$  is a node in element  $e$  (otherwise  $\varphi_i = 0$ )
- The  $d + 1$  nonzero  $b_i^{(e)}$  can be collected in an *element vector*  $\tilde{b}_r^{(e)} = \{\tilde{b}_r^{(e)}\}, r \in I_d$

Assembly:

$$b_{q(e,r)} := b_{q(e,r)} + \tilde{b}_r^{(e)}, \quad r, s \in I_d$$

# Mapping to a reference element

Instead of computing

$$\tilde{A}_{r,s}^{(e)} = \int_{\Omega^{(e)}} \varphi_{q(e,r)}(x) \varphi_{q(e,s)}(x) dx = \int_{x_L}^{x_R} \varphi_{q(e,r)}(x) \varphi_{q(e,s)}(x) dx$$

we now map  $[x_L, x_R]$  to a standardized reference element domain  $[-1, 1]$  with local coordinate  $X$

# Affine mapping

$$x = \frac{1}{2}(x_L + x_R) + \frac{1}{2}(x_R - x_L)X$$

or rewritten as

$$x = x_m + \frac{1}{2}hX, \quad x_m = (x_L + x_R)/2$$

# Integral transformation

Reference element integration: just change integration variable from  $x$  to  $X$ . Introduce local basis function

$$\tilde{\varphi}_r(X) = \varphi_{q(e,r)}(x(X))$$

$$\tilde{A}_{r,s}^{(e)} = \int_{\Omega^{(e)}} \varphi_{q(e,r)}(x) \varphi_{q(e,s)}(x) dx = \int_{-1}^1 \tilde{\varphi}_r(X) \tilde{\varphi}_s(X) \underbrace{\frac{dx}{dX}}_{\det J = h/2} dX = \int_{-1}^1 \tilde{\varphi}_r$$

$$\tilde{b}_r^{(e)} = \int_{\Omega^{(e)}} f(x) \varphi_{q(e,r)}(x) dx = \int_{-1}^1 f(x(X)) \tilde{\varphi}_r(X) \det J dX$$



# Advantages of the reference element

- Always the same domain for integration:  $[-1, 1]$
- We only need formulas for  $\tilde{\varphi}_r(X)$  over one element (no piecewise polynomial definition)
- $\tilde{\varphi}_r(X)$  is the same for all elements: no dependence on element length and location, which is "factored out" in the mapping and  $\det J$

## Standardized basis functions for P1 elements

$$\tilde{\varphi}_0(X) = \frac{1}{2}(1 - X) \quad (8)$$

$$\tilde{\varphi}_1(X) = \frac{1}{2}(1 + X) \quad (9)$$

## Standardized basis functions for P2 elements

P2 elements:

$$\tilde{\varphi}_0(X) = \frac{1}{2}(X - 1)X \quad (10)$$

$$\tilde{\varphi}_1(X) = 1 - X^2 \quad (11)$$

$$\tilde{\varphi}_2(X) = \frac{1}{2}(X + 1)X \quad (12)$$

Easy to generalize to arbitrary order!

## Integration over a reference element; element matrix

P1 elements and  $f(x) = x(1 - x)$ .

$$\begin{aligned}\tilde{A}_{0,0}^{(e)} &= \int_{-1}^1 \tilde{\varphi}_0(X) \tilde{\varphi}_0(X) \frac{h}{2} dX \\ &= \int_{-1}^1 \frac{1}{2}(1 - X) \frac{1}{2}(1 - X) \frac{h}{2} dX = \frac{h}{8} \int_{-1}^1 (1 - X)^2 dX = \frac{h}{3}\end{aligned}\quad (13)$$

$$\begin{aligned}\tilde{A}_{1,0}^{(e)} &= \int_{-1}^1 \tilde{\varphi}_1(X) \tilde{\varphi}_0(X) \frac{h}{2} dX \\ &= \int_{-1}^1 \frac{1}{2}(1 + X) \frac{1}{2}(1 - X) \frac{h}{2} dX = \frac{h}{8} \int_{-1}^1 (1 - X^2) dX = \frac{h}{6}\end{aligned}\quad (14)$$

$$\tilde{A}_{0,1}^{(e)} = \tilde{A}_{1,0}^{(e)} \quad (15)$$

$$\tilde{A}_{1,1}^{(e)} = \int_{-1}^1 \tilde{\varphi}_1(X) \tilde{\varphi}_1(X) \frac{h}{2} dX$$

## Integration over a reference element; element vector

$$\begin{aligned}\tilde{b}_0^{(e)} &= \int_{-1}^1 f(x(X)) \tilde{\varphi}_0(X) \frac{h}{2} dX \\&= \int_{-1}^1 (x_m + \frac{1}{2}hX)(1 - (x_m + \frac{1}{2}hX)) \frac{1}{2}(1 - X) \frac{h}{2} dX \\&= -\frac{1}{24}h^3 + \frac{1}{6}h^2x_m - \frac{1}{12}h^2 - \frac{1}{2}hx_m^2 + \frac{1}{2}hx_m\end{aligned}\quad (17)$$

$$\begin{aligned}\tilde{b}_1^{(e)} &= \int_{-1}^1 f(x(X)) \tilde{\varphi}_1(X) \frac{h}{2} dX \\&= \int_{-1}^1 (x_m + \frac{1}{2}hX)(1 - (x_m + \frac{1}{2}hX)) \frac{1}{2}(1 + X) \frac{h}{2} dX \\&= -\frac{1}{24}h^3 - \frac{1}{6}h^2x_m + \frac{1}{12}h^2 - \frac{1}{2}hx_m^2 + \frac{1}{2}hx_m\end{aligned}\quad (18)$$

$x_m$ : element midpoint.

## Tedious calculations! Let's use symbolic software

```
>>> import sympy as sp
>>> x, x_m, h, X = sp.symbols('x x_m h X')
>>> sp.integrate(h/8*(1-X)**2, (X, -1, 1))
h/3
>>> sp.integrate(h/8*(1+X)*(1-X), (X, -1, 1))
h/6
>>> x = x_m + h/2*X
>>> b_0 = sp.integrate(h/4*x*(1-x)*(1-X), (X, -1, 1))
>>> print b_0
-h**3/24 + h**2*x_m/6 - h**2/12 - h*x_m**2/2 + h*x_m/2
```

Can print out in  $\text{\LaTeX}$  too (convenient for copying into reports):

```
>>> print sp.latex(b_0, mode='plain')
- \frac{1}{24} h^3 + \frac{1}{6} h^2 x_{\text{m}}
- \frac{1}{12} h^2 - \frac{1}{2} h x_{\text{m}}^2
+ \frac{1}{2} h x_{\text{m}}
```

# Implementation

- Coming functions appear in `fe_approx1D.py`
- Functions can operate in symbolic or numeric mode
- The code documents all steps in finite element calculations!

# Compute finite element basis functions in the reference element

Let  $\tilde{\varphi}_r(X)$  be a Lagrange polynomial of degree  $d$ :

```
import sympy as sp
import numpy as np

def phi_r(r, X, d):
    if isinstance(X, sp.Symbol):
        h = sp.Rational(1, d) # node spacing
        nodes = [2*i*h - 1 for i in range(d+1)]
    else:
        # assume X is numeric: use floats for nodes
        nodes = np.linspace(-1, 1, d+1)
    return Lagrange_polynomial(X, r, nodes)

def Lagrange_polynomial(x, i, points):
    p = 1
    for k in range(len(points)):
        if k != i:
            p *= (x - points[k]) / (points[i] - points[k])
    return p

def basis(d=1):
    """Return the complete basis."""
    X = sp.Symbol('X')
    phi = [phi_r(r, X, d) for r in range(d+1)]
    return phi
```



# Compute the element matrix

```
def element_matrix(phi, Omega_e, symbolic=True):
    n = len(phi)
    A_e = sp.zeros((n, n))
    X = sp.Symbol('X')
    if symbolic:
        h = sp.Symbol('h')
    else:
        h = Omega_e[1] - Omega_e[0]
    detJ = h/2 # dx/dX
    for r in range(n):
        for s in range(r, n):
            A_e[r,s] = sp.integrate(phi[r]*phi[s]*detJ, (X, -1, 1))
            A_e[s,r] = A_e[r,s]
    return A_e
```

## Example on symbolic vs numeric element matrix

```
>>> from fe_approx1D import *
>>> phi = basis(d=1)
>>> phi
[1/2 - X/2, 1/2 + X/2]
>>> element_matrix(phi, Omega_e=[0.1, 0.2], symbolic=True)
[h/3, h/6]
[h/6, h/3]
>>> element_matrix(phi, Omega_e=[0.1, 0.2], symbolic=False)
[0.03333333333333333, 0.01666666666666667]
[0.01666666666666667, 0.03333333333333333]
```

# Compute the element vector

```
def element_vector(f, phi, Omega_e, symbolic=True):
    n = len(phi)
    b_e = sp.zeros((n, 1))
    # Make f a function of X
    X = sp.Symbol('X')
    if symbolic:
        h = sp.Symbol('h')
    else:
        h = Omega_e[1] - Omega_e[0]
    x = (Omega_e[0] + Omega_e[1])/2 + h/2*X # mapping
    f = f.subs('x', x) # substitute mapping formula for x
    detJ = h/2 # dx/dX
    for r in range(n):
        b_e[r] = sp.integrate(f*phi[r]*detJ, (X, -1, 1))
    return b_e
```

Note `f.subs('x', x)`: replace `x` by `x(X)` such that `f` contains `X`

# Fallback on numerical integration if symbolic integration fails

- Element matrix: only polynomials and sympy always succeeds
- Element vector:  $\int f \tilde{\varphi} dx$  can fail (sympy then returns an Integral object instead of a number)

```
def element_vector(f, phi, Omega_e, symbolic=True):
    ...
    I = sp.integrate(f*phi[r]*detJ, (X, -1, 1)) # try...
    if isinstance(I, sp.Integral):
        h = Omega_e[1] - Omega_e[0] # Ensure h is numerical
        detJ = h/2
        integrand = sp.lambdify([X], f*phi[r]*detJ)
        I = sp.mpmath.quad(integrand, [-1, 1])
    b_e[r] = I
    ...
```

# Linear system assembly and solution

```
def assemble(nodes, elements, phi, f, symbolic=True):
    N_n, N_e = len(nodes), len(elements)
    zeros = sp.zeros if symbolic else np.zeros
    A = zeros((N_n, N_n))
    b = zeros((N_n, 1))
    for e in range(N_e):
        Omega_e = [nodes[elements[e][0]], nodes[elements[e][-1]]]

        A_e = element_matrix(phi, Omega_e, symbolic)
        b_e = element_vector(f, phi, Omega_e, symbolic)

        for r in range(len(elements[e])):
            for s in range(len(elements[e])):
                A[elements[e][r], elements[e][s]] += A_e[r, s]
            b[elements[e][r]] += b_e[r]
    return A, b
```

# Linear system solution

```
if symbolic:
    c = A.LUsolve(b)           # sympy arrays, symbolic Gaussian el
else:
    c = np.linalg.solve(A, b)  # numpy arrays, numerical solve
```

Note: the symbolic computation of  $A$  and  $b$  and the symbolic solution can be very tedious.

## Example on computing symbolic approximations

```
>>> h, x = sp.symbols('h x')
>>> nodes = [0, h, 2*h]
>>> elements = [[0, 1], [1, 2]]
>>> phi = basis(d=1)
>>> f = x*(1-x)
>>> A, b = assemble(nodes, elements, phi, f, symbolic=True)
>>> A
[h/3,   h/6,   0]
[h/6,  2*h/3, h/6]
[ 0,   h/6, h/3]
>>> b
[      h**2/6 - h**3/12]
[      h**2 - 7*h**3/6]
[5*h**2/6 - 17*h**3/12]
>>> c = A.LUsolve(b)
>>> c
[
                                h**2/6]
[12*(7*h**2/12 - 35*h**3/72)/(7*h)]
[ 7*(4*h**2/7 - 23*h**3/21)/(2*h)]
```

# Example on computing numerical approximations

```
>>> nodes = [0, 0.5, 1]
>>> elements = [[0, 1], [1, 2]]
>>> phi = basis(d=1)
>>> x = sp.Symbol('x')
>>> f = x*(1-x)
>>> A, b = assemble(nodes, elements, phi, f, symbolic=False)
>>> A
[ 0.16666666666666667, 0.08333333333333333, 0]
[0.08333333333333333, 0.3333333333333333, 0.08333333333333333]
[ 0, 0.08333333333333333, 0.16666666666666667]
>>> b
[ 0.03125]
[0.10416666666666667]
[ 0.03125]
>>> c = A.LUsolve(b)
>>> c
[0.04166666666666666]
[ 0.2916666666666667]
[0.04166666666666666]
```



# The structure of the coefficient matrix

```
>>> d=1; N_e=8; Omega=[0,1]  # 8 linear elements on [0,1]
>>> phi = basis(d)
>>> f = x*(1-x)
>>> nodes, elements = mesh_symbolic(N_e, d, Omega)
>>> A, b = assemble(nodes, elements, phi, f, symbolic=True)
>>> A
[h/3,    h/6,    0,    0,    0,    0,    0,    0,    0]
[h/6, 2*h/3,    h/6,    0,    0,    0,    0,    0,    0]
[ 0,    h/6, 2*h/3,    h/6,    0,    0,    0,    0,    0]
[ 0,    0,    h/6, 2*h/3,    h/6,    0,    0,    0,    0]
[ 0,    0,    0,    h/6, 2*h/3,    h/6,    0,    0,    0]
[ 0,    0,    0,    0,    h/6, 2*h/3,    h/6,    0,    0]
[ 0,    0,    0,    0,    0,    h/6, 2*h/3,    h/6,    0]
[ 0,    0,    0,    0,    0,    0,    h/6, 2*h/3,    h/6]
[ 0,    0,    0,    0,    0,    0,    0,    h/6, h/3]
```

Note: do this by hand to understand what is going on!

# General result: the coefficient matrix is sparse

- Sparse = most of the entries are zeros
- Below: P1 elements

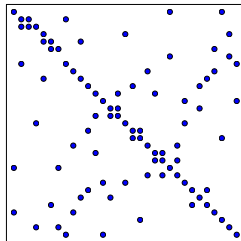
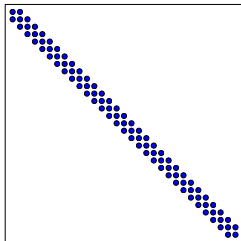
$$A = \frac{h}{6} \begin{pmatrix} 2 & 1 & 0 & \dots & \dots & \dots & \dots & \dots & 0 \\ 1 & 4 & 1 & \ddots & & & & & \vdots \\ 0 & 1 & 4 & 1 & \ddots & & & & \vdots \\ \vdots & \ddots & & \ddots & \ddots & 0 & & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & 0 & 1 & 4 & 1 & \ddots & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & & \ddots & 1 & 4 & 1 \\ 0 & \dots & \dots & \dots & \dots & \dots & 0 & 1 & 2 \end{pmatrix}$$

## Exemplifying the sparsity for P2 elements

$$A = \frac{h}{30} \begin{pmatrix} 4 & 2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 16 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 2 & 8 & 2 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 16 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 2 & 8 & 2 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 16 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 2 & 8 & 2 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 16 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 2 & 4 \end{pmatrix}$$

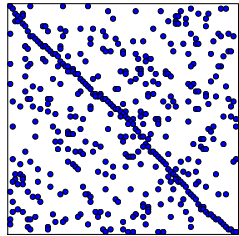
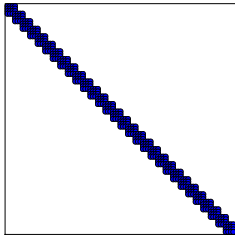
# Matrix sparsity pattern for regular/random numbering of P1 elements

- Left: number nodes and elements from left to right
- Right: number nodes and elements arbitrarily



# Matrix sparsity pattern for regular/random numbering of P3 elements

- Left: number nodes and elements from left to right
- Right: number nodes and elements arbitrarily



# Sparse matrix storage and solution

The minimum storage requirements for the coefficient matrix  $A_{i,j}$ :

- P1 elements: only 3 nonzero entires per row
- P2 elements: only 5 nonzero entires per row
- P3 elements: only 7 nonzero entires per row
- It is important to utilize sparse storage and sparse solvers
- In Python: `scipy.sparse` package

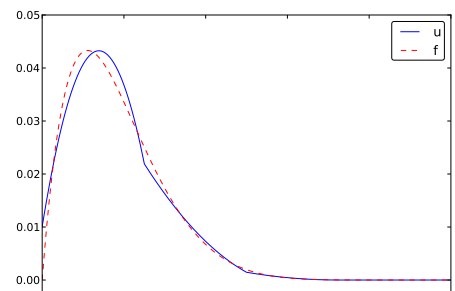
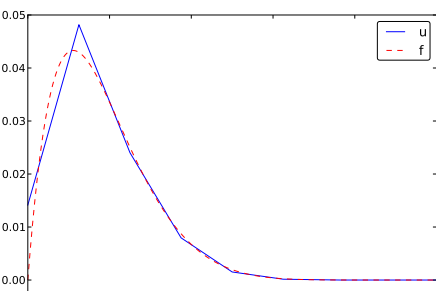
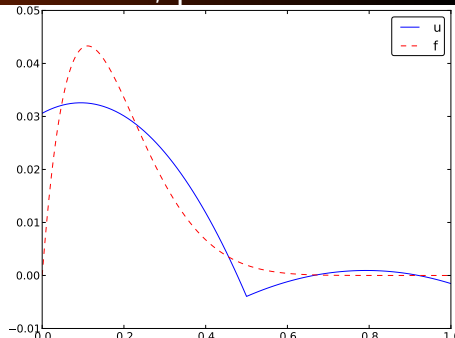
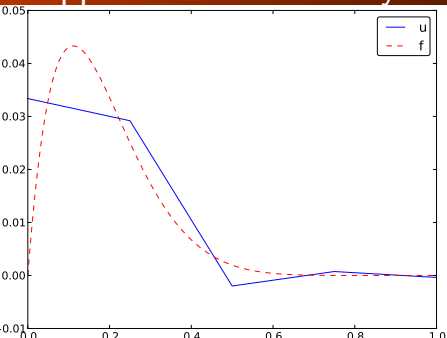
# Approximate $f \sim x^9$ by various elements; code

Compute a mesh with  $N_e$  elements, basis functions of degree  $d$ , and approximate a given symbolic expression  $f(x)$  by a finite element expansion  $u(x) = \sum_j c_j \varphi_j(x)$ :

```
import sympy as sp
from fe_approx1D import approximate
x = sp.Symbol('x')

approximate(f=x*(1-x)**8, symbolic=False, d=1, N_e=4)
approximate(f=x*(1-x)**8, symbolic=False, d=2, N_e=2)
approximate(f=x*(1-x)**8, symbolic=False, d=1, N_e=8)
approximate(f=x*(1-x)**8, symbolic=False, d=2, N_e=4)
```

# Approximate $f \sim x^9$ by various elements; plot





# Comparison of finite element and finite difference approximation

- Finite difference approximation of a function  $f(x)$ : simply choose  $u_i = f(x_i)$  (interpolation)
- Galerkin/projection and least squares method: must derive and solve a linear system
- What is *really* the difference in  $u$ ?

# Interpolation/collocation with finite elements

Let  $\{x_i\}_{i \in \mathcal{I}_s}$  be the nodes in the mesh. Collocation means

$$u(x_i) = f(x_i), \quad i \in \mathcal{I}_s,$$

which translates to

$$\sum_{j \in \mathcal{I}_s} c_j \varphi_j(x_i) = f(x_i),$$

but  $\varphi_j(x_i) = 0$  if  $i \neq j$  so the sum collapses to one term  $c_i \varphi_i(x_i) = c_i$ , and we have the result

$$c_i = f(x_i)$$

Same result as the standard finite difference approach, but finite elements define  $u$  also *between* the  $x_i$  points

# Galerkin/project and least squares vs collocation/interpolation or finite differences

- Scope: work with P1 elements
- Use projection/Galerkin or least squares (equivalent)
- Interpret the resulting linear system as finite difference equations

The P1 finite element machinery results in a linear system where equation no  $i$  is

$$\frac{h}{6}(u_{i-1} + 4u_i + u_{i+1}) = (f, \varphi_i)$$

Note:

- We have used  $u_i$  for  $c_i$  to make notation similar to finite differences
- The finite difference counterpart is just  $u_i = f_i$

## Expressing the left-hand side in finite difference operator notation

Rewrite the left-hand side of finite element equation no  $i$ :

$$h(u_i + \frac{1}{6}(u_{i-1} - 2u_i + u_{i+1})) = [h(u + \frac{h^2}{6}D_x D_x u)]_i$$

This is the standard finite difference approximation of

$$h(u + \frac{h^2}{6}u'')$$

## Treating the right-hand side; Trapezoidal rule

$$(f, \varphi_i) = \int_{x_{i-1}}^{x_i} f(x) \frac{1}{h} (x - x_{i-1}) dx + \int_{x_i}^{x_{i+1}} f(x) \frac{1}{h} (1 - (x - x_i)) dx$$

Cannot do much unless we specialize  $f$  or use *numerical integration*.

Trapezoidal rule using the nodes:

$$(f, \varphi_i) = \int_{\Omega} f \varphi_i dx \approx h \frac{1}{2} (f(x_0) \varphi_i(x_0) + f(x_N) \varphi_i(x_N)) + h \sum_{j=1}^{N-1} f(x_j) \varphi_i(x_j)$$

$\varphi_i(x_j) = \delta_{ij}$ , so this formula collapses to one term:

$$(f, \varphi_i) \approx hf(x_i), \quad i = 1, \dots, N-1.$$

Same result as in collocation (interpolation) and the finite difference method!

## Treating the right-hand side; Simpson's rule

$$\int_{\Omega} g(x) dx \approx \frac{h}{6} \left( g(x_0) + 2 \sum_{j=1}^{N-1} g(x_j) + 4 \sum_{j=0}^{N-1} g(x_{j+\frac{1}{2}}) + f(x_{2N}) \right),$$

Our case:  $g = f\varphi_i$ . The sums collapse because  $\varphi_i = 0$  at most of the points.

$$(f, \varphi_i) \approx \frac{h}{3} (f_{i-\frac{1}{2}} + f_i + f_{i+\frac{1}{2}})$$

Conclusions:

- While the finite difference method just samples  $f$  at  $x_i$ , the finite element method applies an average (smoothing) of  $f$  around  $x_i$
- On the left-hand side we have a term  $\sim hu''$ , and  $u''$  also contribute to smoothing
- There is some inherent smoothing in the finite element method

# Finite element approximation vs finite differences

With Trapezoidal integration of  $(f, \varphi_i)$ , the finite element method essentially solve

$$u + \frac{h^2}{6} u'' = f, \quad u'(0) = u'(L) = 0,$$

by the finite difference method

$$\left[ u + \frac{h^2}{6} D_x D_x u = f \right]_i$$

With Simpson integration of  $(f, \varphi_i)$  we essentially solve

$$\left[ u + \frac{h^2}{6} D_x D_x u = \bar{f} \right]_i,$$

where

$$\bar{f}_i = \frac{1}{3} (f_{i-1/2} + f_i + f_{i+1/2})$$

Note: as  $h \rightarrow 0$ ,  $hu'' \rightarrow 0$  and  $\bar{f}_i \rightarrow f_i$ .

# Making finite elements behave as finite differences

- Can we adjust the finite element method so that we do not get the extra  $hu''$  smoothing term and averaging of  $f$ ?
- This is sometimes important in time-dependent problems to incorporate good properties of finite differences into finite elements

Result:

- Compute all integrals by the Trapezoidal method and P1 elements
- Specifically, the coefficient matrix becomes diagonal ("lumped") - no linear system (!)
- Loss of accuracy? The Trapezoidal rule has error  $\mathcal{O}(h^2)$ , the same as the approximation error in P1 elements



# Limitations of the nodes and element concepts

So far,

- *Nodes*: points for defining  $\varphi_i$  and computing  $u$  values
- *Elements*: subdomain (containing a few nodes)
- This is a common notion of nodes and elements

One problem:

- Our algorithms need nodes at the element boundaries
- This is often not desirable, so we need to throw the `nodes` and `elements` arrays away and find a more generalized element concept

# A generalized element concept

- We introduce *cell* for the subdomain that we up to now called element
- A cell has *vertices* (interval end points)
- *Nodes* are, almost as before, points where we want to compute unknown functions
- *Degrees of freedom* is what the  $c_j$  represent (usually function values at nodes)

# The concept of a finite element

- 1 a *reference cell* in a local reference coordinate system
- 2 a set of *basis functions*  $\tilde{\varphi}_r$  defined on the cell
- 3 a set of *degrees of freedom* (e.g., function values) that uniquely determine the basis functions such that  $\tilde{\varphi}_r = 1$  for degree of freedom number  $r$  and  $\tilde{\varphi}_r = 0$  for all other degrees of freedom
- 4 a mapping between local and global degree of freedom numbers (*dof map*)
- 5 a geometric *mapping* of the reference cell onto to cell in the physical domain:  $[-1, 1] \Rightarrow [x_L, x_R]$

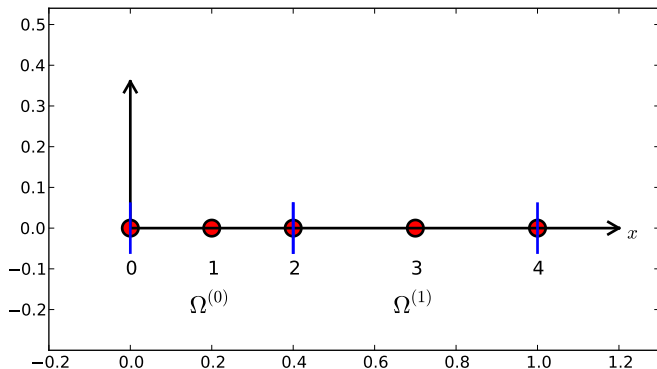
## Implementation; basic data structures

- Cell vertex coordinates: `vertices` (equals nodes for P1 elements)
- Element vertices: `cell[e][r]` holds global vertex number of local vertex no `r` in element `e` (same as `elements` for P1 elements)
- `dof_map[e,r]` maps local dof `r` in element `e` to global dof number (same as `elements` for P $d$  elements)

The assembly process now applies `dof_map`:

```
A[dof_map[e][r], dof_map[e][s]] += A_e[r,s]  
b[dof_map[e][r]] += b_e[r]
```

# Implementation; example with P2 elements



```
vertices = [0, 0.4, 1]
cells = [[0, 1], [1, 2]]
dof_map = [[0, 1, 2], [2, 3, 4]]
```

## Implementation; example with P0 elements

Example: Same mesh, but  $u$  is piecewise constant in each cell (P0 element). Same vertices and cells, but

```
dof_map = [[0], [1]]
```

May think of one node in the middle of each element.

We will hereafter work with cells, vertices, and dof\_map.

# Example on doing the algorithmic steps

```
# Use modified fe_approx1D module
from fe_approx1D_numint import *

x = sp.Symbol('x')
f = x*(1 - x)

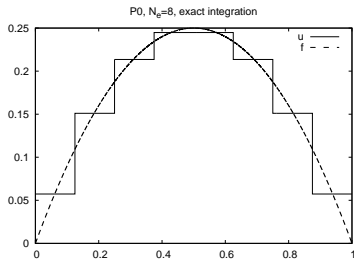
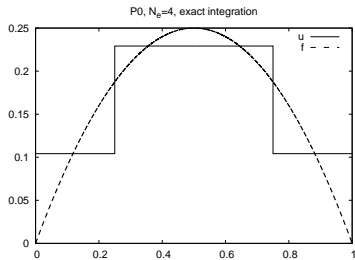
N_e = 10
# Create mesh with P3 (cubic) elements
vertices, cells, dof_map = mesh_uniform(N_e, d=3, Omega=[0,1])

# Create basis functions on the mesh
phi = [basis(len(dof_map[e])-1) for e in range(N_e)]

# Create linear system and solve it
A, b = assemble(vertices, cells, dof_map, phi, f)
c = np.linalg.solve(A, b)

# Make very fine mesh and sample u(x) on this mesh for plotting
x_u, u = u_glob(c, vertices, cells, dof_map,
                 resolution_per_element=51)
plot(x_u, u)
```

# Approximating a parabola by P0 elements



The approximate function automates the steps in the previous slide:

```
from fe_approx1D_numint import *  
x=sp.Symbol("x")  
for N_e in 4, 8:  
    approximate(x*(1-x), d=0, N_e=N_e, Omega=[0,1])
```



# Computing the error of the approximation; principles

$$L^2 \text{ error: } \|e\|_{L^2} = \left( \int_{\Omega} e^2 dx \right)^{1/2}$$

Accurate approximation of the integral:

- Sample  $u(x)$  at many points in each element (call `u_glob`, returns  $x$  and  $u$ )
- Use the Trapezoidal rule based on the samples
- It is important to integrate  $u$  accurately *over the elements*
- (In a finite difference method we would just sample the mesh point values)

# Computing the error of the approximation; details

## Note.

We need a version of the Trapezoidal rule valid for non-uniformly spaced points:

$$\int_{\Omega} g(x) dx \approx \sum_{j=0}^{n-1} \frac{1}{2} (g(x_j) + g(x_{j+1})) (x_{j+1} - x_j)$$

```
# Given c, compute x and u values on a very fine mesh
x, u = u_glob(c, vertices, cells, dof_map,
               resolution_per_element=101)
# Compute the error on the very fine mesh
e = f(x) - u
e2 = e**2
# Vectorized Trapezoidal rule
E = np.sqrt(0.5*np.sum((e2[:-1] + e2[1:]))*(x[1:] - x[:-1])))
```

## How does the error depend on $h$ and $d$ ?

Theory and experiments show that the least squares or projection/Galerkin method in combination with  $P^d$  elements of equal length  $h$  has an error

$$\|e\|_{L^2} = Ch^{d+1}$$

where  $C$  depends on  $f$ , but not on  $h$  or  $d$ .

# Cubic Hermite polynomials; definition

- Can we construct  $\varphi_i(x)$  with continuous derivatives? Yes!

Consider a reference cell  $[-1, 1]$ . We introduce two nodes,  $X = -1$  and  $X = 1$ . The degrees of freedom are

- 0: value of function at  $X = -1$
- 1: value of first derivative at  $X = -1$
- 2: value of function at  $X = 1$
- 3: value of first derivative at  $X = 1$

Derivatives as unknowns ensure the same  $\varphi'_i(x)$  value at nodes and thereby continuous derivatives.

# Cubic Hermite polynomials; derivation

4 constraints on  $\tilde{\varphi}_r$  (1 for dof  $r$ , 0 for all others):

- $\tilde{\varphi}_0(X_{(0)}) = 1, \tilde{\varphi}_0(X_{(1)}) = 0, \tilde{\varphi}'_0(X_{(0)}) = 0, \tilde{\varphi}'_0(X_{(1)}) = 0$
- $\tilde{\varphi}'_1(X_{(0)}) = 1, \tilde{\varphi}'_1(X_{(1)}) = 0, \tilde{\varphi}_1(X_{(0)}) = 0, \tilde{\varphi}_1(X_{(1)}) = 0$
- $\tilde{\varphi}_2(X_{(1)}) = 1, \tilde{\varphi}_2(X_{(0)}) = 0, \tilde{\varphi}'_2(X_{(0)}) = 0, \tilde{\varphi}'_2(X_{(1)}) = 0$
- $\tilde{\varphi}'_3(X_{(1)}) = 1, \tilde{\varphi}'_3(X_{(0)}) = 0, \tilde{\varphi}_3(X_{(0)}) = 0, \tilde{\varphi}_3(X_{(1)}) = 0$

This gives 4 linear, coupled equations *for each*  $\tilde{\varphi}_r$  to determine the 4 coefficients in the cubic polynomial

$$\tilde{\varphi}_0(X) = 1 - \frac{3}{4}(X+1)^2 + \frac{1}{4}(X+1)^3 \quad (19)$$

$$\tilde{\varphi}_1(X) = -(X+1)\left(1 - \frac{1}{2}(X+1)\right)^2 \quad (20)$$

$$\tilde{\varphi}_2(X) = \frac{3}{4}(X+1)^2 - \frac{1}{2}(X+1)^3 \quad (21)$$

$$\tilde{\varphi}_3(X) = -\frac{1}{2}(X+1)\left(\frac{1}{2}(X+1)^2 - (X+1)\right) \quad (22)$$

$$(23)$$

# Numerical integration

- $\int_{\Omega} f \varphi_i dx$  must in general be computed by numerical integration
- Numerical integration is often used for the matrix too

Common form of a numerical integration rule:

$$\int_{-1}^1 g(X) dX \approx \sum_{j=0}^M w_j g(\bar{X}_j),$$

where

- $\bar{X}_j$  are *integration points*
- $w_j$  are *integration weights*

Different rules correspond to different choices of points and weights

# The Midpoint rule

Simplest possibility: the Midpoint rule,

$$\int_{-1}^1 g(X) dX \approx 2g(0), \quad \bar{X}_0 = 0, \quad w_0 = 2,$$

Exact for linear integrands



# Newton-Cotes rules

- Idea: use a fixed, uniformly distributed set of points in  $[-1, 1]$
- The points often coincides with nodes
- Very useful for making  $\varphi_i \varphi_j = 0$  and get diagonal ("mass") matrices ("lumping")

The Trapezoidal rule:

$$\int_{-1}^1 g(X) dX \approx g(-1) + g(1), \quad \bar{X}_0 = -1, \bar{X}_1 = 1, w_0 = w_1 = 1,$$

Simpson's rule:

$$\int_{-1}^1 g(X) dX \approx \frac{1}{3} (g(-1) + 4g(0) + g(1)),$$

where

$$\bar{X}_0 = -1, \bar{X}_1 = 0, \bar{X}_2 = 1, w_0 = w_2 = \frac{1}{3}, w_1 = \frac{4}{3}$$

# Gauss-Legendre rules with optimized points

- Optimize the location of points to get higher accuracy
- Gauss-Legendre rules (quadrature) adjust points and weights to integrate polynomials exactly

$$M = 1: \quad \bar{X}_0 = -\frac{1}{\sqrt{3}}, \quad \bar{X}_1 = \frac{1}{\sqrt{3}}, \quad w_0 = w_1 = 1 \quad (24)$$

$$M = 2: \quad \bar{X}_0 = -\sqrt{\frac{3}{5}}, \quad \bar{X}_1 = 0, \quad \bar{X}_2 = \sqrt{\frac{3}{5}}, \quad w_0 = w_2 = \frac{5}{9}, \quad w_1 = \frac{8}{9} \quad (25)$$

- $M = 1$ : integrates 3rd degree polynomials exactly
- $M = 2$ : integrates 5th degree polynomials exactly
- In general,  $M$ -point rule integrates a polynomial of degree  $2M + 1$  exactly.

See [numint.py](#) for a large collection of Gauss-Legendre rules.

# Approximation of functions in 2D

## Extensibility of 1D ideas.

All the concepts and algorithms developed for approximation of 1D functions  $f(x)$  can readily be extended to 2D functions  $f(x, y)$  and 3D functions  $f(x, y, z)$ . Key formulas stay the same.

Inner product in 2D:

$$(f, g) = \int_{\Omega} f(x, y)g(x, y)dx dy$$

Least squares and project/Galerkin lead to a linear system

$$\sum_{j \in \mathcal{I}_s} A_{i,j} c_j = b_i, \quad i \in \mathcal{I}_s$$

$$A_{i,j} = (\psi_i, \psi_j)$$

$$b_i = (f, \psi_i)$$

## 2D basis functions as tensor products of 1D functions

Use a 1D basis for  $x$  variation and a similar for  $y$  variation:

$$V_x = \text{span}\{\hat{\psi}_0(x), \dots, \hat{\psi}_{N_x}(x)\} \quad (26)$$

$$V_y = \text{span}\{\hat{\psi}_0(y), \dots, \hat{\psi}_{N_y}(y)\} \quad (27)$$

The 2D vector space can be defined as a *tensor product*  $V = V_x \otimes V_y$  with basis functions

$$\psi_{p,q}(x, y) = \hat{\psi}_p(x) \hat{\psi}_q(y) \quad p \in \mathcal{I}_x, q \in \mathcal{I}_y.$$

# Tensor products

Given two vectors  $a = (a_0, \dots, a_M)$  and  $b = (b_0, \dots, b_N)$  their *outer tensor product*, also called the *dyadic product*, is  $p = a \otimes b$ , defined through

$$p_{i,j} = a_i b_j, \quad i = 0, \dots, M, \quad j = 0, \dots, N.$$

Note:  $p$  has two indices (as a matrix or two-dimensional array)

Example: 2D basis as tensor product of 1D spaces,

$$\psi_{p,q}(x, y) = \hat{\psi}_p(x) \hat{\psi}_q(y), \quad p \in \mathcal{I}_x, q \in \mathcal{I}_y$$

## Double or single index?

The 2D basis can employ a double index and double sum:

$$u = \sum_{p \in \mathcal{I}_x} \sum_{q \in \mathcal{I}_y} c_{p,q} \psi_{p,q}(x, y)$$

Or just a single index:

$$u = \sum_{j \in \mathcal{I}_s} c_j \psi_j(x, y)$$

with

$$\psi_i(x, y) = \hat{\psi}_p(x) \hat{\psi}_q(y), \quad i = pN_y + q \text{ or } i = qN_x + p$$

## Example on 2D (bilinear) basis functions; formulas

In 1D we use the basis

$$\{1, x\}$$

2D tensor product (all combinations):

$$\psi_{0,0} = 1, \quad \psi_{1,0} = x, \quad \psi_{0,1} = y, \quad \psi_{1,1} = xy$$

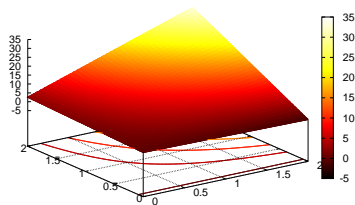
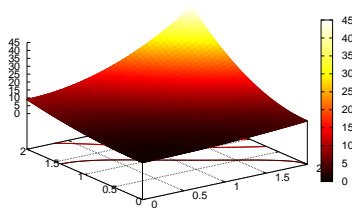
or with a single index:

$$\psi_0 = 1, \quad \psi_1 = x, \quad \psi_2 = y, \quad \psi_3 = xy$$

See notes for details of a hand-calculation.

# Example on 2D (bilinear) basis functions; plot

Quadratic  $f(x, y) = (1 + x^2)(1 + 2y^2)$  (left), bilinear  $u$  (right):





## Implementation; principal changes to the 1D code

Very small modification of `approx1D.py`:

- $\Omega = [[0, L_x], [0, L_y]]$
- Symbolic integration in 2D
- Construction of 2D (tensor product) basis functions

# Implementation; 2D integration

```
import sympy as sp

integrand = psi[i]*psi[j]
I = sp.integrate(integrand,
                 (x, Omega[0][0], Omega[0][1]),
                 (y, Omega[1][0], Omega[1][1]))

# Fall back on numerical integration if symbolic integration
# was unsuccessful
if isinstance(I, sp.Integral):
    integrand = sp.lambdify([x,y], integrand)
    I = sp.mpmath.quad(integrand,
                       [Omega[0][0], Omega[0][1]],
                       [Omega[1][0], Omega[1][1]])
```

# Implementation; 2D basis functions

Tensor product of 1D "Taylor-style" polynomials  $x^i$ :

```
def taylor(x, y, Nx, Ny):  
    return [x**i*y**j for i in range(Nx+1) for j in range(Ny+1)]
```

Tensor product of 1D sine functions  $\sin((i+1)\pi x)$ :

```
def sines(x, y, Nx, Ny):  
    return [sp.sin(sp.pi*(i+1)*x)*sp.sin(sp.pi*(j+1)*y)  
            for i in range(Nx+1) for j in range(Ny+1)]
```

Complete code in [approx2D.py](#)

$$f(x, y) = (1 + x^2)(1 + 2y^2)$$

```
>>> from approx2D import *
>>> f = (1+x**2)*(1+2*y**2)
>>> psi = taylor(x, y, 1, 1)
>>> Omega = [[0, 2], [0, 2]]
>>> u, c = least_squares(f, psi, Omega)
>>> print u
8*x*y - 2*x/3 + 4*y/3 - 1/9
>>> print sp.expand(f)
2*x**2*y**2 + x**2 + 2*y**2 + 1
```

## Implementation; trying a perfect expansion

Add higher powers to the basis such that  $f \in V$ :

```
>>> psi = taylor(x, y, 2, 2)
>>> u, c = least_squares(f, psi, Omega)
>>> print u
2*x**2*y**2 + x**2 + 2*y**2 + 1
>>> print u-f
0
```

Expected:  $u = f$  when  $f \in V$

# Generalization to 3D

Key idea:

$$V = V_x \otimes V_y \otimes V_z$$

Repeated outer tensor product of multiple vectors.

$$a^{(q)} = (a_0^{(q)}, \dots, a_{N_q}^{(q)}), \quad q = 0, \dots, m$$

$$p = a^{(0)} \otimes \dots \otimes a^{(m)}$$

$$p_{i_0, i_1, \dots, i_m} = a_{i_0}^{(0)} a_{i_1}^{(1)} \dots a_{i_m}^{(m)}$$

$$\psi_{p,q,r}(x, y, z) = \hat{\psi}_p(x) \hat{\psi}_q(y) \hat{\psi}_r(z)$$

$$u(x, y, z) = \sum_{p \in \mathcal{I}_x} \sum_{q \in \mathcal{I}_y} \sum_{r \in \mathcal{I}_z} c_{p,q,r} \psi_{p,q,r}(x, y, z)$$

The two great advantages of the finite element method:

- Can handle complex-shaped domains in 2D and 3D
- Can easily provide higher-order polynomials in the approximation

Finite elements in 1D: mostly for learning, insight, debugging

# Examples on cell types

2D:

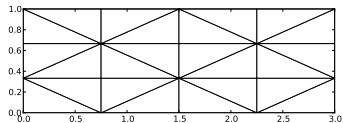
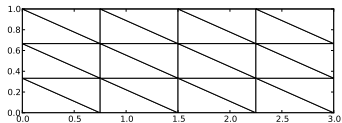
- triangles
- quadrilaterals

3D:

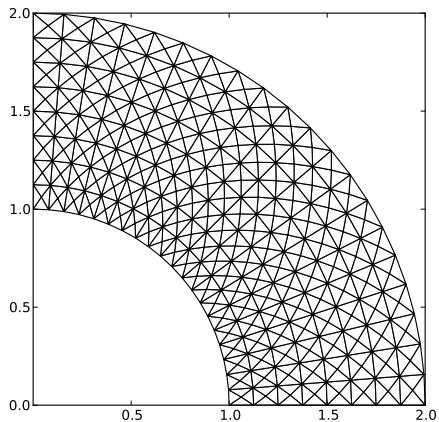
- tetrahedra
- hexahedra



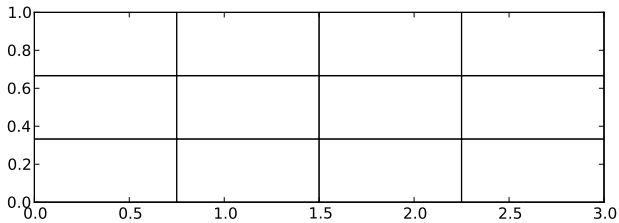
# Rectangular domain with 2D P1 elements



# Deformed geometry with 2D P1 elements

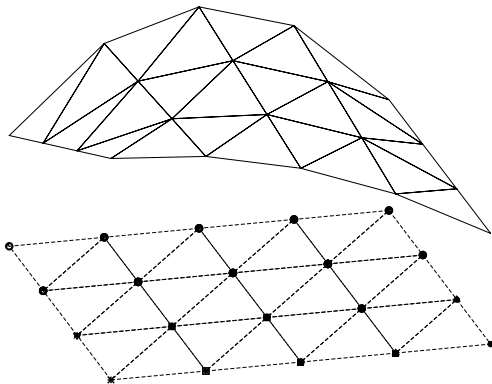


# Rectangular domain with 2D Q1 elements



# Basis functions over triangles in the physical domain

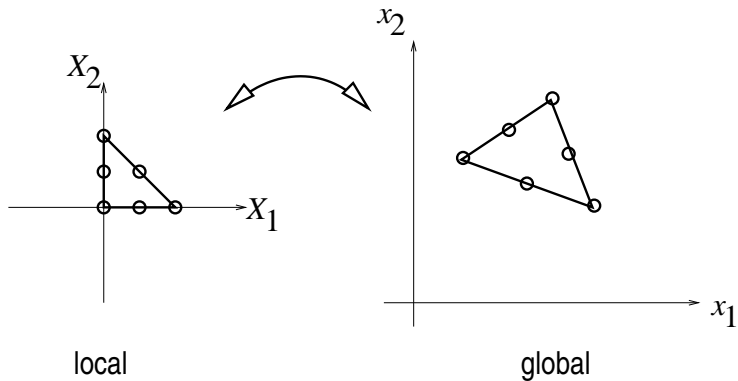
The P1 triangular 2D element:  $u$  is linear  $ax + by + c$  over each triangular cell



# Basic features of 2D P1 elements

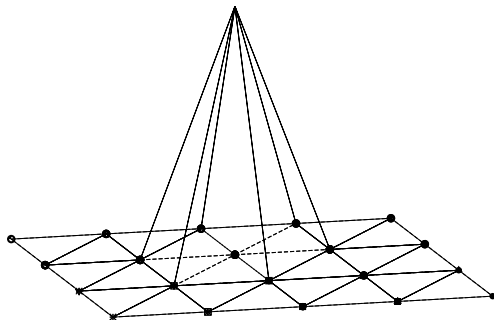
- $\varphi_r(X, Y)$  is a linear function over each element
- Cells = triangles
- Vertices = corners of the cells
- Nodes = vertices
- Degrees of freedom = function values at the nodes

# Linear mapping of reference element onto general triangular cell



## $\varphi_i$ : pyramid shape, composed of planes

- $\varphi_i(X, Y)$  varies linearly over an element
- $\varphi_i = 1$  at vertex (node)  $i$ , 0 at all other vertices (nodes)

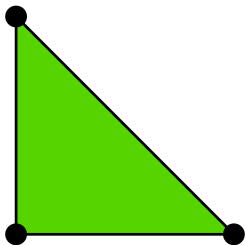


# Element matrices and vectors

- As in 1D, the contribution from one cell to the matrix involves just a few numbers, collected in the element matrix and vector
- $\varphi_i \varphi_j \neq 0$  only if  $i$  and  $j$  are degrees of freedom (vertices/nodes) in the same element
- The 2D P1 has a  $3 \times 3$  element matrix



## Basis functions over triangles in the reference cell



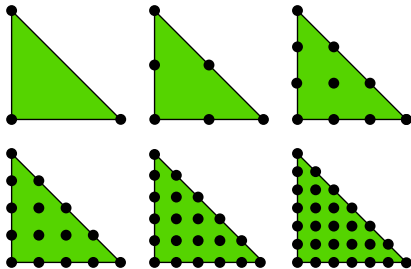
$$\tilde{\varphi}_0(X, Y) = 1 - X - Y \quad (28)$$

$$\tilde{\varphi}_1(X, Y) = X \quad (29)$$

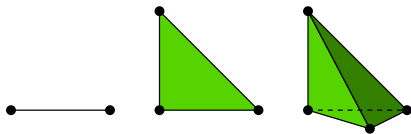
$$\tilde{\varphi}_2(X, Y) = Y \quad (30)$$

Higher-degree  $\tilde{\varphi}_r$  introduce more nodes (dof = node values)

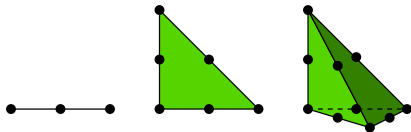
## 2D P1, P2, P3, P4, P5, and P6 elements



# P1 elements in 1D, 2D, and 3D



## P2 elements in 1D, 2D, and 3D



- Interval, triangle, tetrahedron: *simplex* element (plural quick-form: *simplices*)
- Side of the cell is called *face*
- Tetrahedron has also *edges*

## Affine mapping of the reference cell; formula

Mapping of local  $\mathbf{X} = (X, Y)$  coordinates in the reference cell to global, physical  $\mathbf{x} = (x, y)$  coordinates:

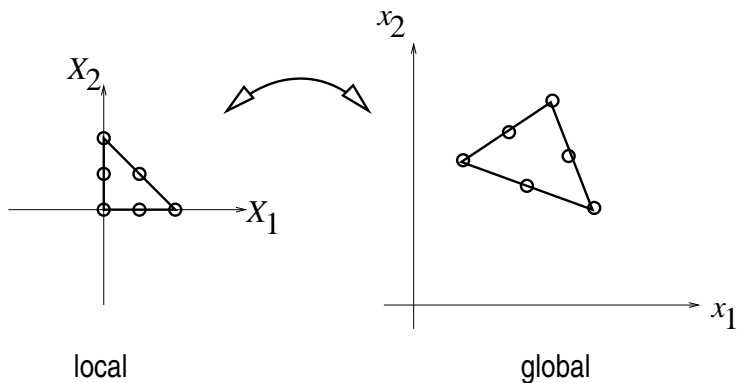
$$\mathbf{x} = \sum_r \tilde{\varphi}_r^{(1)}(\mathbf{X}) \mathbf{x}_{q(e,r)} \quad (31)$$

where

- $r$  runs over the local vertex numbers in the cell
- $\mathbf{x}_i$  are the  $(x, y)$  coordinates of vertex  $i$
- $\tilde{\varphi}_r^{(1)}$  are P1 basis functions

This mapping preserves the straight/planar faces and edges.

# Affine mapping of the reference cell; figure

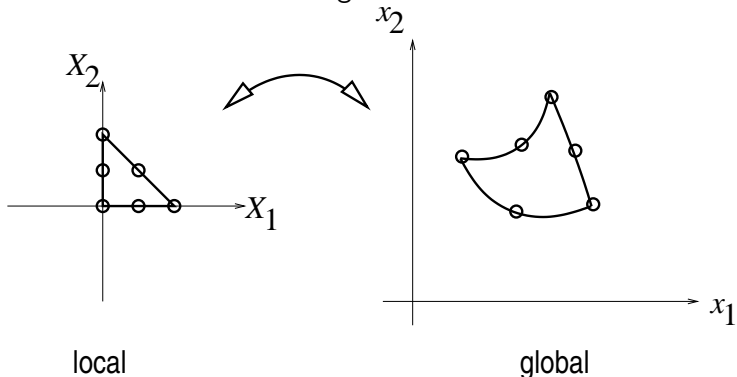


# Isoparametric mapping of the reference cell

Idea: Use the basis functions of the element (not only the P1 functions) to map the element

$$\mathbf{x} = \sum_r \tilde{\varphi}_r(\mathbf{X}) \mathbf{x}_{q(e,r)}$$

Advantage: higher-order polynomial basis functions now map the reference cell to a *curved* triangle or tetrahedron.



# Computing integrals

Integrals must be transformed from  $\Omega^{(e)}$  (physical cell) to  $\tilde{\Omega}^r$  (reference cell):

$$\int_{\Omega^{(e)}} \varphi_i(\mathbf{x}) \varphi_j(\mathbf{x}) \, d\mathbf{x} = \int_{\tilde{\Omega}^r} \tilde{\varphi}_i(\mathbf{X}) \tilde{\varphi}_j(\mathbf{X}) \det J \, d\mathbf{X} \quad (32)$$

$$\int_{\Omega^{(e)}} \varphi_i(\mathbf{x}) f(\mathbf{x}) \, d\mathbf{x} = \int_{\tilde{\Omega}^r} \tilde{\varphi}_i(\mathbf{X}) f(\mathbf{x}(\mathbf{X})) \det J \, d\mathbf{X} \quad (33)$$

where  $d\mathbf{x} = dx dy$  or  $d\mathbf{x} = dx dy dz$  and  $\det J$  is the determinant of the Jacobian of the mapping  $\mathbf{x}(\mathbf{X})$ .

$$J = \begin{bmatrix} \frac{\partial x}{\partial X} & \frac{\partial x}{\partial Y} \\ \frac{\partial y}{\partial X} & \frac{\partial y}{\partial Y} \end{bmatrix}, \quad \det J = \frac{\partial x}{\partial X} \frac{\partial y}{\partial Y} - \frac{\partial x}{\partial Y} \frac{\partial y}{\partial X}$$

Affine mapping (31):  $\det J = 2\Delta$ ,  $\Delta$  = cell volume

!slide **Remark on going from 1D to 2D/3D**

Finite elements in 2D and 3D builds on the same *ideas* and



# Differential equation models

Our aim is to extend the ideas for approximating  $f$  by  $u$ , or solving

$$u = f$$

to real differential equations like[[[

$$-u'' + bu = f, \quad u(0) = 1, \quad u'(L) = D$$

Three methods are addressed:

- 1 least squares
- 2 Galerkin/projection
- 3 collocation (interpolation)

Method 2 will be totally dominating!

$$\mathcal{L}(u) = 0, \quad x \in \Omega \quad (34)$$

Examples (1D problems):

$$\mathcal{L}(u) = \frac{d^2 u}{dx^2} - f(x), \quad (35)$$

$$\mathcal{L}(u) = \frac{d}{dx} \left( \alpha(x) \frac{du}{dx} \right) + f(x), \quad (36)$$

$$\mathcal{L}(u) = \frac{d}{dx} \left( \alpha(u) \frac{du}{dx} \right) - au + f(x), \quad (37)$$

$$\mathcal{L}(u) = \frac{d}{dx} \left( \alpha(u) \frac{du}{dx} \right) + f(u, x) \quad (38)$$

# Abstract boundary conditions

$$\mathcal{B}_0(u) = 0, \quad x = 0, \quad \mathcal{B}_1(u) = 0, \quad x = L \quad (39)$$

Examples:

$$\mathcal{B}_i(u) = u - g, \quad \text{Dirichlet condition} \quad (40)$$

$$\mathcal{B}_i(u) = -\alpha \frac{du}{dx} - g, \quad \text{Neumann condition} \quad (41)$$

$$\mathcal{B}_i(u) = -\alpha \frac{du}{dx} - h(u - g), \quad \text{Robin condition} \quad (42)$$

## Reminder about notation

- $u_e(x)$  is the symbol for the *exact* solution of  $\mathcal{L}(u_e) = 0$
- $u(x)$  denotes an *approximate* solution
- We seek  $u \in V$
- $V = \text{span}\{\psi_0(x), \dots, \psi_N(x)\}$ ,  $V$  has basis  $\{\psi_i\}_{i \in \mathcal{I}_s}$
- $\mathcal{I}_s = \{0, \dots, N\}$  is an index set
- $u(x) = \sum_{j \in \mathcal{I}_s} c_j \psi_j(x)$
- Inner product:  $(u, v) = \int_{\Omega} uv \, dx$
- Norm:  $\|u\| = \sqrt{(u, u)}$

Much is similar to approximating a function (solving  $u = f$ ), but two new topics are needed:

- Variational formulation of the differential equation problem (including integration by parts)
- Handling of boundary conditions

# Residual-minimizing principles

- When solving  $u = f$  we knew the error  $e = f - u$  and could use principles for minimizing the error
- When solving  $\mathcal{L}(u_e) = 0$  we do not know  $u_e$  and cannot work with the error  $e = u_e - u$
- We only have the *error in the equation*: the residual  $R$

Inserting  $u = \sum_j c_j \psi_j$  in  $\mathcal{L} = 0$  gives a residual

$$R = \mathcal{L}(u) = \mathcal{L}\left(\sum_j c_j \psi_j\right) \neq 0 \quad (43)$$

Goal: minimize  $R$  wrt  $\{c_i\}_{i \in \mathcal{I}_s}$  (and hope it makes a small  $e$  too)

$$R = R(c_0, \dots, c_N; x)$$

# The least squares method

Idea: minimize

$$E = \|R\|^2 = (R, R) = \int_{\Omega} R^2 dx \quad (44)$$

Minimization wrt  $\{c_i\}_{i \in \mathcal{I}_s}$  implies

$$\frac{\partial E}{\partial c_i} = \int_{\Omega} 2R \frac{\partial R}{\partial c_i} dx = 0 \quad \Leftrightarrow \quad (R, \frac{\partial R}{\partial c_i}) = 0, \quad i \in \mathcal{I}_s \quad (45)$$

$N + 1$  equations for  $N + 1$  unknowns  $\{c_i\}_{i \in \mathcal{I}_s}$

# The Galerkin method

Idea: make  $R$  orthogonal to  $V$ ,

$$(R, v) = 0, \quad \forall v \in V \quad (46)$$

This implies

$$(R, \psi_i) = 0, \quad i \in \mathcal{I}_s \quad (47)$$

$N + 1$  equations for  $N + 1$  unknowns  $\{c_i\}_{i \in \mathcal{I}_s}$



# The Method of Weighted Residuals

Generalization of the Galerkin method: demand  $R$  orthogonal to some space  $W$ , possibly  $W \neq V$ :

$$(R, v) = 0, \quad \forall v \in W \quad (48)$$

If  $\{w_0, \dots, w_N\}$  is a basis for  $W$ :

$$(R, w_i) = 0, \quad i \in \mathcal{I}_s \quad (49)$$

- $N + 1$  equations for  $N + 1$  unknowns  $\{c_i\}_{i \in \mathcal{I}_s}$
- Weighted residual with  $w_i = \partial R / \partial c_i$  gives least squares

# Terminology: test and trial Functions

- $\psi_j$  used in  $\sum_j c_j \psi_j$  is called *trial function*
- $\psi_i$  or  $w_i$  used as weight in Galerkin's method is called *test function*

# The collocation method

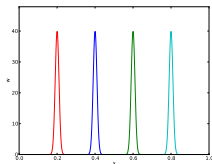
Idea: demand  $R = 0$  at  $N + 1$  points

$$R(x_i; c_0, \dots, c_N) = 0, \quad i \in \mathcal{I}_s \quad (50)$$

Note: The collocation method is a weighted residual method with delta functions as weights

$$0 = \int_{\Omega} R(x; c_0, \dots, c_N) \delta(x - x_i) dx = R(x_i; c_0, \dots, c_N)$$

property of  $\delta(x)$  :  $\int_{\Omega} f(x) \delta(x - x_i) dx = f(x_i), \quad x_i \in \Omega \quad (51)$



# Examples on using the principles

## Goal.

Exemplify the least squares, Galerkin, and collocation methods in a simple 1D problem with global basis functions.

# The first model problem

$$-u''(x) = f(x), \quad x \in \Omega = [0, L], \quad u(0) = 0, \quad u(L) = 0 \quad (52)$$

Basis functions:

$$\psi_i(x) = \sin \left( (i+1) \pi \frac{x}{L} \right), \quad i \in \mathcal{I}_s \quad (53)$$

The residual:

$$\begin{aligned} R(x; c_0, \dots, c_N) &= u''(x) + f(x), \\ &= \frac{d^2}{dx^2} \left( \sum_{j \in \mathcal{I}_s} c_j \psi_j(x) \right) + f(x), \\ &= - \sum_{j \in \mathcal{I}_s} c_j \psi_j''(x) + f(x) \end{aligned} \quad (54)$$

# Boundary conditions

Since  $u(0) = u(L) = 0$  we must ensure that all  $\psi_i(0) = \psi_i(L) = 0$ .  
Then

$$u(0) = \sum_j c_j \psi_j(0) = 0, \quad u(L) = \sum_j c_j \psi_j(L)$$

- $u$  known: Dirichlet boundary condition
- $u'$  known: Neumann boundary condition
- Must have  $\psi_i = 0$  where Dirichlet conditions apply

# The least squares method; principle

$$(R, \frac{\partial R}{\partial c_i}) = 0, \quad i \in \mathcal{I}_s$$

$$\frac{\partial R}{\partial c_i} = \frac{\partial}{\partial c_i} \left( \sum_{j \in \mathcal{I}_s} c_j \psi_j''(x) + f(x) \right) = \psi_i''(x) \quad (55)$$

Because:

$$\frac{\partial}{\partial c_i} (c_0 \psi_0'' + c_1 \psi_1'' + \cdots + c_{i-1} \psi_{i-1}'' + c_i \psi_i'' + c_{i+1} \psi_{i+1}'' + \cdots + c_N \psi_N'') =$$

## The least squares method; equation system

$$\left(\sum_j c_j \psi_j'' + f, \psi_i''\right) = 0, \quad i \in \mathcal{I}_s \quad (56)$$

Rearrangement:

$$\sum_{j \in \mathcal{I}_s} (\psi_i'', \psi_j'') c_j = -(f, \psi_i''), \quad i \in \mathcal{I}_s \quad (57)$$

This is a linear system

$$\sum_{j \in \mathcal{I}_s} A_{i,j} c_j = b_i, \quad i \in \mathcal{I}_s$$

with

$$\begin{aligned} A_{i,j} &= (\psi_i'', \psi_j'') \\ &= \pi^4 (i+1)^2 (j+1)^2 L^{-4} \int_0^L \sin\left((i+1)\pi \frac{x}{L}\right) \sin\left((j+1)\pi \frac{x}{L}\right) dx \end{aligned}$$



# Orthogonality of the basis functions gives diagonal matrix

Useful property:

$$\int_0^L \sin\left((i+1)\pi\frac{x}{L}\right) \sin\left((j+1)\pi\frac{x}{L}\right) dx = \delta_{ij}, \quad \delta_{ij} = \begin{cases} \frac{1}{2}L & i=j \\ 0, & i \neq j \end{cases} \quad (60)$$

$\Rightarrow (\psi_i'', \psi_j'') = \delta_{ij}$ , i.e., diagonal  $A_{i,j}$ , and we can easily solve for  $c_i$ :

$$c_i = \frac{2L}{\pi^2(i+1)^2} \int_0^L f(x) \sin\left((i+1)\pi\frac{x}{L}\right) dx \quad (61)$$

# Least squares method; solution

Let's sympy do the work ( $f(x) = 2$ ):

```
from sympy import *
import sys

i, j = symbols('i j', integer=True)
x, L = symbols('x L')
f = 2
a = 2*L/(pi**2*(i+1)**2)
c_i = a*integrate(f*sin((i+1)*pi*x/L), (x, 0, L))
c_i = simplify(c_i)
print c_i
```

$$c_i = 4 \frac{L^2 \left( (-1)^i + 1 \right)}{\pi^3 (i^3 + 3i^2 + 3i + 1)}, \quad u(x) = \sum_{k=0}^{N/2} \frac{8L^2}{\pi^3 (2k+1)^3} \sin \left( (2k+1) \pi \frac{x}{L} \right) \quad (62)$$

Fast decay:  $c_2 = c_0/27$ ,  $c_4 = c_0/125$  - only one term might be good enough:

$$\left( \frac{8L^2}{\pi^3} \right) \sin \left( \frac{x}{L} \right)$$

# The Galerkin method; principle

$$R = u'' + f:$$

$$(u'' + f, v) = 0, \quad \forall v \in V,$$

or

$$(u'', v) = -(f, v), \quad \forall v \in V \quad (63)$$

This is a *variational formulation* of the differential equation problem.

$\forall v \in V$  means for all basis functions:

$$\left( \sum_{j \in \mathcal{I}_s} c_j \psi_j'', \psi_i \right) = -(f, \psi_i), \quad i \in \mathcal{I}_s \quad (64)$$

# The Galerkin method; solution

Since  $\psi_i'' \propto \psi_i$ , Galerkin's method gives the same linear system and the same solution as the least squares method (in this particular example).

# The collocation method

$R = 0$  (i.e., the differential equation) must be satisfied at  $N + 1$  points:

$$-\sum_{j \in \mathcal{I}_s} c_j \psi_j''(x_i) = f(x_i), \quad i \in \mathcal{I}_s \quad (65)$$

This is a linear system  $\sum_j A_{i,j} = b_i$  with entries

$$A_{i,j} = -\psi_j''(x_i) = (j+1)^2 \pi^2 L^{-2} \sin\left((j+1)\pi \frac{x_i}{L}\right), \quad b_i = 2$$

Choose:  $N = 0$ ,  $x_0 = L/2$

$$c_0 = 2L^2/\pi^2$$

# Comparison of the methods

- Exact solution:  $u(x) = x(L - x)$
- Galerkin or least squares ( $N = 0$ ):  $u(x) = 8L^2\pi^{-3} \sin(\pi x/L)$
- Collocation method ( $N = 0$ ):  $u(x) = 2L^2\pi^{-2} \sin(\pi x/L)$ .
- Max error in Galerkin/least sq.:  $-0.008L^2$
- Max error in collocation:  $0.047L^2$

# Integration by parts

Second-order derivatives will hereafter be integrated by parts

$$\begin{aligned}\int_0^L u''(x)v(x)dx &= - \int_0^L u'(x)v'(x)dx + [vu']_0^L \\ &= - \int_0^L u'(x)v'(x)dx + u'(L)v(L) - u'(0)v(0)\end{aligned}\tag{66}$$

Motivation:

- Lowers the order of derivatives
- Gives more symmetric forms (incl. matrices)
- Enables easy handling of Neumann boundary conditions
- Finite element basis functions  $\varphi_i$  have discontinuous derivatives (at cell boundaries) and are not suited for terms with  $\varphi_i''$

**Boundary function; principles**

## Boundary function; example (1)

Dirichlet conditions:  $u(0) = C$  and  $u(L) = D$ . Choose for example

$$B(x) = \frac{1}{L}(C(L-x) + Dx) : \quad B(0) = C, \quad B(L) = D$$

$$u(x) = B(x) + \sum_{j \in \mathcal{I}_s} c_j \psi_j(x), \quad (67)$$

$$u(0) = B(0) = C, \quad u(L) = B(L) = D$$



## Boundary function; example (2)

Dirichlet condition:  $u(L) = D$ . Choose for example

$$B(x) = D : \quad B(L) = D$$

$$u(x) = B(x) + \sum_{j \in \mathcal{I}_s} c_j \psi_j(x), \quad (68)$$

$$u(L) = B(L) = D$$

# Impact of the boundary function on the space where we seek the solution

- $\{\psi_i\}_{i \in \mathcal{I}_s}$  is a basis for  $V$
- $\sum_{j \in \mathcal{I}_s} c_j \psi_j(x) \in V$
- But  $u \notin V$ !
- Reason: say  $u(0) = C$  and  $u \in V$  (any  $v \in V$  has  $v(0) = C$ , then  $2u \notin V$  because  $2u(0) = 2C$
- When  $u(x) = B(x) + \sum_{j \in \mathcal{I}_s} c_j \psi_j(x)$ ,  $B \neq 0$ ,  $B \notin V$  (in general) and  $u \notin V$ , but  $(u - B) \in V$  since  $\sum_j c_j \psi_j \in V$

# Abstract notation for variational formulations

The finite element literature (and much FEniCS documentation) applies an abstract notation for the variational formulation:

Find  $(u - B) \in V$  such that

$$a(u, v) = L(v) \quad \forall v \in V$$

## Example on abstract notation

$$-u'' = f, \quad u'(0) = C, \quad u(L) = D, \quad u = D + \sum_j c_j \psi_j$$

Variational formulation:

$$\int_{\Omega} u' v' dx = \int_{\Omega} f v dx - v(0)C \quad \text{or} \quad (u', v') = (f, v) - v(0)C \quad \forall v \in V$$

Abstract formulation: find  $(u - B) \in V$  such that

$$a(u, v) = L(v) \quad \forall v \in V$$

We identify

$$a(u, v) = (u', v'), \quad L(v) = (f, v) - v(0)C$$

# Bilinear and linear forms

- $a(u, v)$  is a *bilinear form*
- $L(v)$  is a *linear form*

Linear form means

$$L(\alpha_1 v_1 + \alpha_2 v_2) = \alpha_1 L(v_1) + \alpha_2 L(v_2),$$

Bilinear form means

$$\begin{aligned}a(\alpha_1 u_1 + \alpha_2 u_2, v) &= \alpha_1 a(u_1, v) + \alpha_2 a(u_2, v), \\a(u, \alpha_1 v_1 + \alpha_2 v_2) &= \alpha_1 a(u, v_1) + \alpha_2 a(u, v_2)\end{aligned}$$

In nonlinear problems: Find  $(u - B) \in V$  such that  
 $F(u; v) = 0 \quad \forall v \in V$

# The linear system associated with abstract form

$$a(u, v) = L(v) \quad \forall v \in V \quad \Leftrightarrow \quad a(u, \psi_i) = L(\psi_i) \quad i \in \mathcal{I}_s$$

We can now derive the corresponding linear system once and for all:

$$a\left(\sum_{j \in \mathcal{I}_s} c_j \psi_j, \psi_i\right) c_j = L(\psi_i) \quad i \in \mathcal{I}_s$$

Because of linearity,

$$\sum_{j \in \mathcal{I}_s} \underbrace{a(\psi_j, \psi_i)}_{A_{i,j}} c_j = \underbrace{L(\psi_i)}_{b_i} \quad i \in \mathcal{I}_s$$

Given  $a(u, v)$  and  $L(v)$  in a problem, we can immediately generate the linear system:

$$A_{i,j} = a(\psi_j, \psi_i), \quad b_i = L(\psi_i)$$

# Equivalence with minimization problem

If  $a(u, v) = a(v, u)$ ,

$$a(u, v) = L(v) \quad \forall v \in V,$$

is equivalent to minimizing the functional

$$F(v) = \frac{1}{2}a(v, v) - L(v)$$

over all functions  $v \in V$ . That is,

$$F(u) \leq F(v) \quad \forall v \in V.$$

- Much used in the early days of finite elements
- Still much used in structural analysis and elasticity
- Not as general as Galerkin's method (since  $a(u, v) = a(v, u)$ )

# Examples on variational formulations

## Goal.

Derive variational formulations for many prototype differential equations in 1D that include

- variable coefficients
- mixed Dirichlet and Neumann conditions
- nonlinear coefficients



## Variable coefficient; problem

$$-\frac{d}{dx} \left( \alpha(x) \frac{du}{dx} \right) = f(x), \quad x \in \Omega = [0, L], \quad u(0) = C, \quad u(L) = D \quad (69)$$

- Variable coefficient  $\alpha(x)$
- *Nonzero* Dirichlet conditions at  $x = 0$  and  $x = L$
- Must have  $\psi_i(0) = \psi_i(L) = 0$
- $V = \text{span}\{\psi_0, \dots, \psi_N\}$
- $v \in V$ :  $v(0) = v(L) = 0$

$$u(x) = B(x) + \sum_{j \in \mathcal{I}_s} c_j \psi_j(x)$$

$$B(x) = C + \frac{1}{L}(D - C)x$$

## Variable coefficient; variational formulation (1)

$$R = -\frac{d}{dx} \left( a \frac{du}{dx} \right) - f$$

Galerkin's method:

$$(R, v) = 0, \quad \forall v \in V,$$

or with integrals:

$$\int_{\Omega} \left( \frac{d}{dx} \left( \alpha \frac{du}{dx} \right) - f \right) v \, dx = 0, \quad \forall v \in V.$$

## Variable coefficient; variational formulation (2)

Integration by parts:

$$-\int_{\Omega} \frac{d}{dx} \left( \alpha(x) \frac{du}{dx} \right) v \, dx = \int_{\Omega} \alpha(x) \frac{du}{dx} \frac{dv}{dx} \, dx - \left[ \alpha \frac{du}{dx} v \right]_0^L.$$

Boundary terms vanish since  $v(0) = v(L) = 0$

**Variational formulation.**

Find  $(u - B) \in V$  such that

$$\int_{\Omega} \alpha(x) \frac{du}{dx} \frac{dv}{dx} \, dx = \int_{\Omega} f(x) v \, dx, \quad \forall v \in V,$$

Compact notation:

$$\underbrace{(\alpha u', v')}_{a(u, v)} = \underbrace{(f, v)}_{L(v)}, \quad \forall v \in V$$

## Variable coefficient; linear system (the easy way)

With

$$a(u, v) = (\alpha u', v), \quad L(v) = (f, v)$$

we can just use the formula for the linear system:

$$A_{i,j} = a(\psi_j, \psi_i) = (\alpha \psi_j', \psi_i') = \int_{\Omega} \alpha \psi_j' \psi_i' \, dx = \int_{\Omega} \psi_i' \alpha \psi_j' \, dx = a(\psi_i, \psi_j) =$$

$$b_i = (f, \psi_i) = \int_{\Omega} f \psi_i \, dx$$

# Variable coefficient; linear system (full derivation)

$$v = \psi_i \text{ and } u = B + \sum_j c_j \psi_j:$$

$$(\alpha B' + \alpha \sum_{j \in \mathcal{I}_s} c_j \psi'_j, \psi'_i) = (f, \psi_i), \quad i \in \mathcal{I}_s.$$

Reorder to form linear system:

$$\sum_{j \in \mathcal{I}_s} (\alpha \psi'_j, \psi'_i) c_j = (f, \psi_i) + (a(D - C)L^{-1}, \psi'_i), \quad i \in \mathcal{I}_s.$$

This is  $\sum_j A_{i,j} c_j = b_i$  with

$$A_{i,j} = (a \psi'_j, \psi'_i) = \int_{\Omega} \alpha(x) \psi'_j(x) \psi'_i(x) \, dx$$

$$b_i = (f, \psi_i) + (a(D - C)L^{-1}, \psi'_i) = \int_{\Omega} \left( f(x) \psi_i(x) + \alpha(x) \frac{D - C}{L} \psi'_i(x) \right) \, dx$$

## First-order derivative in the equation and boundary condition; problem

$$-u''(x) + bu'(x) = f(x), \quad x \in \Omega = [0, L], \quad u(0) = C, \quad u'(L) = E \quad (70)$$

New features:

- first-order derivative  $u'$  in the equation
- boundary condition with  $u'$ :  $u'(L) = E$

Initial steps:

- Must force  $\psi_i(0) = 0$  because of Dirichlet condition at  $x = 0$
- Boundary function:  $B(x) = C(L - x)$  or just  $B(x) = C$
- No requirements on  $\psi_i(L)$  (no Dirichlet condition at  $x = L$ )

## First-order derivative in the equation and boundary condition; details

$$u = C + \sum_{j \in \mathcal{I}_s} c_j \psi_j(x)$$

Galerkin's method: multiply by  $v$ , integrate over  $\Omega$ , integrate by parts.

$$(-u'' + bu' - f, v) = 0, \quad \forall v \in V$$

$$(u', v') + (bu', v) = (f, v) + [u'v]_0^L, \quad \forall v \in V$$

Now,  $[u'v]_0^L = u'(L)v(L) = Ev(L)$  because  $v(0) = 0$  and  $u'(L) = E$ :

$$(u'v') + (bu', v) = (f, v) + Ev(L), \quad \forall v \in V$$

## First-order derivative in the equation and boundary condition; observations

$$(u'v') + (bu', v) = (f, v) + Ev(L), \quad \forall v \in V,$$

Important:

- The boundary term can be used to implement Neumann conditions
- Forgetting the boundary term implies the condition  $u' = 0$  (!)
- Such conditions are called *natural boundary conditions*



# First-order derivative in the equation and boundary condition; abstract notation

Abstract notation:

$$a(u, v) = L(v) \quad \forall v \in V$$

Here:

$$\begin{aligned} a(u, v) &= (u', v') + (bu', v) \\ L(v) &= (f, v) + Ev(L) \end{aligned}$$

# First-order derivative in the equation and boundary condition; linear system

Insert  $u = C + \sum_j c_j \psi_j$  and  $v = \psi_i$ :

$$\sum_{j \in \mathcal{I}_s} \underbrace{((\psi'_j, \psi'_i) + (b\psi'_j, \psi_i))}_{A_{i,j}} c_j = \underbrace{(f, \psi_i) + E\psi_i(L)}_{b_i}$$

Observation:  $A_{i,j}$  is not symmetric because of the term

$$(b\psi'_j, \psi_i) = \int_{\Omega} b\psi'_j \psi_i dx \neq \int_{\Omega} b\psi'_i \psi_j dx = (\psi'_i, b\psi_j)$$

## Terminology: natural and essential boundary conditions

$$(u', v') + (bu', v) = (f, v) + u'(L)v(L) - u'(0)v(0)$$

- Note: forgetting the boundary terms implies  $u'(L) = u'(0) = 0$  (unless prescribe a Dirichlet condition)
- Conditions on  $u'$  are simply inserted in the variational form and called *natural conditions*
- Conditions on  $u$  at  $x = 0$  requires modifying  $V$  (through  $\psi_i(0) = 0$ ) and are known as *essential conditions*

### Lesson learned.

It is easy to forget the boundary term when integrating by parts. That mistake may prescribe a condition on  $u'$ !

Problem:

$$-(\alpha(u)u')' = f(u), \quad x \in [0, L], \quad u(0) = 0, \quad u'(L) = E \quad (71)$$

- $V$ : basis  $\{\psi_i\}_{i \in \mathcal{I}_s}$  with  $\psi_i(0) = 0$  because of  $u(0) = 0$
- How does the nonlinear coefficients  $\alpha(u)$  and  $f(u)$  impact the variational formulation?
- (Not much!)

# Nonlinear coefficient; variational formulation

Galerkin: multiply by  $v$ , integrate, integrate by parts

$$\int_0^L \alpha(u) \frac{du}{dx} \frac{dv}{dx} dx = \int_0^L f(u) v dx + [\alpha(u) v u']_0^L \quad \forall v \in V$$

- $\alpha(u(0))v(0)u'(0) = 0$  since  $v(0) = 0$
- $\alpha(u(L))v(L)u'(L) = \alpha(u(L))v(L)E$  since  $u'(L) = E$

$$\int_0^L \alpha(u) \frac{du}{dx} \frac{dv}{dx} dx = \int_0^L f(u) v dx + \alpha(u(L))v(L)E \quad \forall v \in V$$

or

$$(\alpha(u)u', v') = (f(u), v) + \alpha(u(L))v(L)E \quad \forall v \in V$$

# Nonlinear coefficient; where does the nonlinearity cause challenges?

- Abstract notation: no  $a(u, v)$  and  $L(v)$  because  $a$  and  $L$  are nonlinear
- Instead:  $F(u; v) = 0 \quad \forall v \in V$
- What about forming a linear system? We get a *nonlinear* system of algebraic equations
- Must use methods like Picard iteration or Newton's method to solve nonlinear algebraic equations
- But: the variational formulation was not much affected by nonlinearities

# Computing with Dirichlet and Neumann conditions; problem

$$-u''(x) = f(x), \quad x \in \Omega = [0, 1], \quad u'(0) = C, \quad u(1) = D$$

- Use a *global* polynomial basis  $\psi_i \sim x^i$  on  $[0, 1]$
- Because of  $u(1) = D$ :  $\psi_i(1) = 0$
- Basis:  $\psi_i(x) = (1 - x)^{i+1}$ ,  $i \in \mathcal{I}_s$
- $B(x) = Dx$

# Computing with Dirichlet and Neumann conditions; details

$$A_{i,j} = (\psi'_j, \psi'_i) = \int_0^1 \psi'_i(x) \psi'_j(x) dx = \int_0^1 (i+1)(j+1)(1-x)^{i+j} dx,$$

Choose  $f(x) = 2$ :

$$\begin{aligned} b_i &= (2, \psi_i) - (D, \psi'_i) - C\psi_i(0) \\ &= \int_0^1 (2(1-x)^{i+1} - D(i+1)(1-x)^i) dx - C\psi_i(0) \end{aligned}$$

Can easily do the integrals with sympy.  $N = 1$ :

$$\begin{pmatrix} 1 & 1 \\ 1 & 4/3 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} -C + D + 1 \\ 2/3 - C + D \end{pmatrix}$$

$$c_0 = -C + D + 2, \quad c_1 = -1,$$



## When the numerical method is exact

Assume that apart from boundary conditions,  $u_e$  lies in the same space  $V$  as where we seek  $u$ :

$$u = B + F, \quad F \in V \quad a(B + F, v) = L(v) \quad \forall v \in V \quad u_e = B + E, \quad E \in V$$

Subtract:  $a(F - E, v) = 0 \Rightarrow E = F$  and  $u = u_e$

## Tasks:

- Address the model problem  $-u''(x) = 2$ ,  $u(0) = u(L) = 0$
- Uniform finite element mesh with P1 elements
- Show all finite element computations in detail

# Variational formulation, finite element mesh, and basis

$$-u''(x) = 2, \quad x \in (0, L), \quad u(0) = u(L) = 0,$$

Variational formulation:

$$(u', v') = (2, v) \quad \forall v \in V$$

Since  $u(0) = 0$  and  $u(L) = 0$ , we must force

$$v(0) = v(L) = 0, \quad \psi_i(0) = \psi_i(L) = 0$$

Use finite element basis, but exclude  $\varphi_0$  and  $\varphi_{N_n}$  since these are not 0 on the boundary:

$$\psi_i = \varphi_{i+1}, \quad i = 0, \dots, N = N_n - 2$$

Introduce index mapping  $\nu(j)$ :  $\psi_i = \varphi_{\nu(i)}$

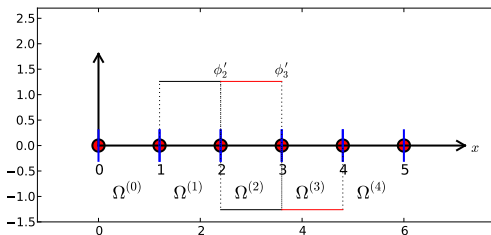
## Computation in the global physical domain; formulas

$$A_{i,j} = \int_0^L \varphi'_{i+1}(x) \varphi'_{j+1}(x) dx, \quad b_i = \int_0^L 2\varphi_{i+1}(x) dx$$

Many will prefer to change indices to obtain a  $\varphi'_i \varphi'_j$  product:  
 $i+1 \rightarrow i, j+1 \rightarrow j$

$$A_{i-1,j-1} = \int_0^L \varphi'_i(x) \varphi'_j(x) dx, \quad b_{i-1} = \int_0^L 2\varphi_i(x) dx$$

# Computation in the global physical domain; details



$$\varphi_i = \pm h^{-1}$$

$$A_{i-1,i-1} = h^{-2}2h = 2h^{-1}, \quad A_{i-1,i-2} = h^{-1}(-h^{-1})h = -h^{-1}, \quad A_{i-1,i} =$$

$$b_{i-1} = 2\left(\frac{1}{2}h + \frac{1}{2}h\right) = 2h$$



# Comparison with a finite difference discretization

- Recall:  $c_i = u(x_{i+1}) \equiv u_{i+1}$
- Write out a general equation at node  $i - 1$ , expressed by  $u_i$

$$-\frac{1}{h}u_{i-1} + \frac{2}{h}u_i - \frac{1}{h}u_{i+1} = 2h \quad (73)$$

The standard finite difference method for  $-u'' = 2$  is

$$-\frac{1}{h^2}u_{i-1} + \frac{2}{h^2}u_i - \frac{1}{h^2}u_{i+1} = 2$$

The finite element method and the finite difference method are identical *in this example*.

(Remains to study the equations involving boundary values)

## Cellwise computations; formulas

- Repeat the previous example, but apply the cellwise algorithm
- Work with one cell at a time
- Transform physical cell to reference cell  $X \in [-1, 1]$

$$A_{i-1,j-1}^{(e)} = \int_{\Omega^{(e)}} \varphi'_i(x) \varphi'_j(x) dx = \int_{-1}^1 \frac{d}{dX} \tilde{\varphi}_r(X) \frac{d}{dX} \tilde{\varphi}_s(X) \frac{h}{2} dX,$$

$$\tilde{\varphi}_0(X) = \frac{1}{2}(1 - X), \quad \tilde{\varphi}_1(X) = \frac{1}{2}(1 + X)$$

$$\frac{d\tilde{\varphi}_0}{dX} = -\frac{1}{2}, \quad \frac{d\tilde{\varphi}_1}{dX} = \frac{1}{2}$$

From the chain rule

$$\frac{d\tilde{\varphi}_r}{dx} = \frac{d\tilde{\varphi}_r}{dX} \frac{dX}{dx} = \frac{2}{h} \frac{d\tilde{\varphi}_r}{dX}$$



## Cellwise computations; details

$$A_{i-1,j-1}^{(e)} = \int_{\Omega^{(e)}} \varphi_i'(x) \varphi_j'(x) dx = \int_{-1}^1 \frac{2}{h} \frac{d\tilde{\varphi}_r}{dX} \frac{2}{h} \frac{d\tilde{\varphi}_s}{dX} \frac{h}{2} dX = \tilde{A}_{r,s}^{(e)}$$

$$b_{i-1}^{(e)} = \int_{\Omega^{(e)}} 2\varphi_i(x) dx = \int_{-1}^1 2\tilde{\varphi}_r(X) \frac{h}{2} dX = \tilde{b}_r^{(e)}, \quad i = q(e, r), \quad r = 0, 1$$

Must run through all  $r, s = 0, 1$  and  $r = 0, 1$  and compute each entry in the element matrix and vector:

$$\tilde{A}^{(e)} = \frac{1}{h} \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}, \quad \tilde{b}^{(e)} = h \begin{pmatrix} 1 \\ 1 \end{pmatrix}. \quad (74)$$

Example:

$$\tilde{A}_{0,1}^{(e)} = \int_{-1}^1 \frac{2}{h} \frac{d\tilde{\varphi}_0}{dX} \frac{2}{h} \frac{d\tilde{\varphi}_1}{dX} \frac{h}{2} dX = \frac{2}{h} \left(-\frac{1}{2}\right) \frac{2}{h} \frac{1}{2} \frac{h}{2} \int_{-1}^1 dX = -\frac{1}{h}$$

## Cellwise computations; details of boundary cells

- The boundary cells involve only one unknown
- $\Omega^{(0)}$ : left node value known, only a contribution from right node
- $\Omega^{(N_e)}$ : right node value known, only a contribution from left node

For  $e = 0$  and  $e = N_e$ :

$$\tilde{A}^{(e)} = \frac{1}{h} \begin{pmatrix} 1 \end{pmatrix}, \quad \tilde{b}^{(e)} = h \begin{pmatrix} 1 \end{pmatrix}$$

Only one degree of freedom ("node") in these cells ( $r = 0$  counts the only dof)

# Cellwise computations; assembly

4 P1 elements:

```
vertices = [0, 0.5, 1, 1.5, 2]
cells = [[0, 1], [1, 2], [2, 3], [3, 4]]
dof_map = [[0], [0, 1], [1, 2], [2]]           # only 1 dof in elm 0, 3
```

Python code for the assembly algorithm:

```
# Ae[e][r,s]: element matrix, be[e][r]: element vector
# A[i,j]: coefficient matrix, b[i]: right-hand side

for e in range(len(Ae)):
    for r in range(Ae[e].shape[0]):
        for s in range(Ae[e].shape[1]):
            A[dof_map[e,r],dof_map[e,s]] += Ae[e][i,j]
            b[dof_map[e,r]] += be[e][i,j]
```

Result: same linear system as arose from computations in the physical domain

# General construction of a boundary function

- Now we address nonzero Dirichlet conditions
- $B(x)$  is not always easy to construct (extend to the interior of  $\Omega$ ), especially not in 2D and 3D
- With finite element  $\varphi_i$ ,  $B(x)$  can be constructed in a completely general way
- $I_b$ : set of indices with nodes where  $u$  is known
- $U_i$ : Dirichlet value of  $u$  at node  $i$ ,  $i \in I_b$

$$B(x) = \sum_{j \in I_b} U_j \varphi_j(x) \quad (75)$$

Suppose we have a Dirichlet condition  $u(x_k) = U_k$ ,  $k \in I_b$ :

$$u(x_k) = \sum_{j \in I_b} U_j \underbrace{\varphi_j(x)}_{\neq 0 \text{ only for } j=k} + \sum_{j \in I_s} c_j \underbrace{\varphi_{\nu(j)}(x_k)}_{=0, k \notin I_s} = U_k$$

## Example with two Dirichlet values; variational formulation

$$-u'' = 2, \quad u(0) = C, \quad u(L) = D$$

$$\int_0^L u' v' \, dx = \int_0^L 2v \, dx \quad \forall v \in V$$

$$(u', v') = (2, v) \quad \forall v \in V$$

## Example with two Dirichlet values; boundary function

$$B(x) = \sum_{j \in I_b} U_j \varphi_j(x) \quad (76)$$

Here  $I_b = \{0, N_n\}$ ,  $U_0 = C$ ,  $U_{N_n} = D$ ,

$$\psi_i = \varphi_{\nu(i)}, \quad \nu(i) = i + 1, \quad i \in \mathcal{I}_s = \{0, \dots, N = N_n - 2\}$$

$$u(x) = C\varphi_0(x) + D\varphi_{N_n}(x) + \sum_{j \in \mathcal{I}_s} c_j \varphi_{\nu(j)} \quad (77)$$

## Example with two Dirichlet values; details

Insert  $u = B + \sum_j c_j \psi_j$  in variational formulation:

$$(u', v') = (2, v) \quad \Rightarrow \quad \left( \sum_j c_j \psi'_j, \psi'_i \right) = (2 - B', \psi_i) \quad \forall v \in V$$

$$\begin{aligned} u(x) &= \underbrace{C \cdot \varphi_0 + D \varphi_{N_n}}_{B(x)} + \sum_{j \in \mathcal{I}_s} c_j \varphi_{j+1} \\ &= C \cdot \varphi_0 + D \varphi_{N_n} + c_0 \varphi_1 + c_1 \varphi_2 + \cdots + c_N \varphi_{N_n-1} \end{aligned}$$

$$A_{i-1,j-1} = \int_0^L \varphi'_i(x) \varphi'_j(x) dx, \quad b_{i-1} = \int_0^L (f(x) - C \varphi'_0(x) - D \varphi'_{N_n}(x)) \varphi_i(x) dx$$

for  $i, j = 1, \dots, N+1 = N_n - 1$ .

New boundary terms from  $-\int B' \varphi_i dx$ :  $C/2$  for  $i = 1$  and  $-D/2$  for  $i = N_n - 1$

## Example with two Dirichlet values; cellwise computations

- Element matrices as in the previous example (with  $u = 0$  on the boundary)
- New element vector in the first and last cell

From the last cell:

$$\tilde{b}_0^{(N_e)} = \int_{-1}^1 \left( f - D \frac{2}{h} \frac{d\tilde{\varphi}_1}{dX} \right) \tilde{\varphi}_0 \frac{h}{2} dX = \left( \frac{h}{2} (2 - D \frac{2}{h} \frac{1}{2}) \right) \int_{-1}^1 \tilde{\varphi}_0 dX = h - D/2$$

From the first cell:

$$\tilde{b}_0^{(0)} = \int_{-1}^1 \left( f - C \frac{2}{h} \frac{d\tilde{\varphi}_0}{dX} \right) \tilde{\varphi}_1 \frac{h}{2} dX = \left( \frac{h}{2} (2 + C \frac{2}{h} \frac{1}{2}) \right) \int_{-1}^1 \tilde{\varphi}_1 dX = h + C/2$$



# Modification of the linear system; ideas

- Method 1: incorporate Dirichlet values through a  $B(x)$  function and demand  $\psi_i = 0$  where Dirichlet values apply
- Method 2: drop  $B(x)$ , drop demands to  $\psi_i$ , just assemble as if there were no Dirichlet conditions, and modify the linear system instead

Method 2: always  $\psi_i = \varphi_i$  and

$$u(x) = \sum_{j \in \mathcal{I}_s} c_j \varphi_j(x), \quad \mathcal{I}_s = \{0, \dots, N = N_n\} \quad (78)$$

Attractive way of incorporating Dirichlet conditions.

$u$  is treated as unknown at all boundaries when computing entires in the linear system

## Modification of the linear system; original system

$$-u'' = 2, \quad u(0) = 0, \quad u(L) = D$$

Assemble as if there were no Dirichlet conditions:

$$\frac{1}{h} \begin{pmatrix} 1 & -1 & 0 & \dots & \dots & \dots & \dots & \dots & 0 \\ -1 & 2 & -1 & \ddots & & & & & \vdots \\ 0 & -1 & 2 & -1 & \ddots & & & & \vdots \\ \vdots & \ddots & & \ddots & \ddots & 0 & & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & 0 & -1 & 2 & -1 & \ddots & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & & \ddots & \ddots & \ddots & -1 \\ 0 & \dots & \dots & \dots & \dots & \dots & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} c_0 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ c_N \end{pmatrix} = \begin{pmatrix} h \\ 2h \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ 2h \\ h \end{pmatrix} \quad (79)$$

## Modification of the linear system; row replacement

- Dirichlet condition  $u(x_k) = U_k$  means  $c_k = U_k$  (since  $c_k = u(x_k)$ )
- Replace first row by  $c_0 = 0$
- Replace last row by  $c_N = D$

$$\frac{1}{h} \begin{pmatrix} h & 0 & 0 & \dots & \dots & \dots & \dots & \dots & 0 \\ -1 & 2 & -1 & \ddots & & & & & \vdots \\ 0 & -1 & 2 & -1 & \ddots & & & & \vdots \\ \vdots & \ddots & & \ddots & \ddots & 0 & & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & 0 & -1 & 2 & -1 & \ddots & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & & \ddots & \ddots & \ddots & -1 \\ 0 & \dots & \dots & \dots & \dots & \dots & 0 & 0 & h \end{pmatrix} \begin{pmatrix} c_0 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ c_N \end{pmatrix} = \begin{pmatrix} 0 \\ 2h \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ 2h \\ D \end{pmatrix} \quad (80)$$

## Modification of the linear system; element matrix/vector

In cell 0 we know  $u$  for local node (degree of freedom)  $r = 0$ .

Replace the first cell equation by  $\tilde{c}_0 = 0$ :

$$\tilde{A}^{(0)} = A = \frac{1}{h} \begin{pmatrix} h & 0 \\ -1 & 1 \end{pmatrix}, \quad \tilde{b}^{(0)} = \begin{pmatrix} 0 \\ h \end{pmatrix} \quad (81)$$

In cell  $N_e$  we know  $u$  for local node  $r = 1$ . Replace the last equation in the cell system by  $\tilde{c}_1 = D$ :

$$\tilde{A}^{(N_e)} = A = \frac{1}{h} \begin{pmatrix} 1 & -1 \\ 0 & h \end{pmatrix}, \quad \tilde{b}^{(N_e)} = \begin{pmatrix} h \\ D \end{pmatrix} \quad (82)$$

# Symmetric modification of the linear system; algorithm

- The modification above destroys symmetry of the matrix:  
e.g.,  $A_{0,1} \neq A_{1,0}$
- Symmetry is often important in 2D and 3D (faster computations)
- A more complex modification can preserve symmetry!

Algorithm for incorporating  $c_i = U_i$  in a symmetric way:

- ➊ Subtract column  $i$  times  $U_i$  from the right-hand side
- ➋ Zero out column and row no  $i$
- ➌ Place 1 on the diagonal
- ➍ Set  $b_i = U_i$



Symmetric modification applied to  $\tilde{A}^{(N_e)}$ :

$$\tilde{A}^{(N_e)} = A = \frac{1}{h} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \tilde{b}^{(N-1)} = \begin{pmatrix} h + D/h \\ D \end{pmatrix} \quad (84)$$

## Neumann conditions.

How can we incorporate  $u'(0) = C$  with finite elements?

$$-u'' = f, \quad u'(0) = C, \quad u(L) = D$$

- $\psi_i(L) = 0$  because of Dirichlet condition  $u(L) = D$
- No demand to  $\psi_i(0)$



# The variational formulation

Galerkin's method:

$$\int_0^L (u''(x) + f(x))\psi_i(x)dx = 0, \quad i \in \mathcal{I}_s$$

Integration of  $u''\psi_i$  by parts:

$$\int_0^L u'(x)\psi'_i(x) dx - (u'(L)\psi_i(L) - u'(0)\psi_i(0)) - \int_0^L f(x)\psi_i(x) dx = 0, \quad i \in \mathcal{I}_s$$

- $u'(L)\psi_i(L) = 0$  since  $\psi_i(L) = 0$
- $u'(0)\psi_i(0) = C\psi_i(0)$  since  $u'(0) = C$

## Method 1: Boundary function and exclusion of Dirichlet degrees of freedom

- $\psi_i = \varphi_i, i \in \mathcal{I}_s = \{0, \dots, N = N_n - 1\}$
- $B(x) = D\varphi_{N_n}(x), u = B + \sum_{j=0}^N c_j \varphi_j$

$$\int_0^L u'(x) \varphi_i'(x) dx = \int_0^L f(x) \varphi_i(x) dx - C \varphi_i(0), \quad i \in \mathcal{I}_s$$

$$\sum_{j=0}^{N=N_n-1} \left( \int_0^L \varphi_i'(x) \varphi_j'(x) dx \right) c_j = \int_0^L (f(x) \varphi_i(x) - D \varphi_N'(x) \varphi_i(x)) dx - C$$

(85)

for  $i = 0, \dots, N = N_n - 1$ .

## Method 2: Use all $\varphi_i$ and insert the Dirichlet condition in the linear system

- Now  $\psi_i = \varphi_i$ ,  $i = 0, \dots, N = N_n$
- $\varphi_N(L) \neq 0$ , so  $u'(L)\varphi_N(L) \neq 0$
- However, the term  $u'(L)\varphi_N(L)$  in  $b_N$  will be erased when we insert the Dirichlet value in  $b_N = D$

We can forget about the term  $u'(L)\varphi_i(L)$ !

### Result.

Boundary terms  $u'\varphi_i$  at points  $x_i$  where Dirichlet values apply can always be forgotten.

$$u(x) = \sum_{j=0}^{N=N_n} c_j \varphi_j(x)$$

$$\sum_{i=0}^{N=N_n} \left( \int_0^L \varphi_i'(x) \varphi_j'(x) dx \right) c_j = \int_0^L f(x) \varphi_i(x) \varphi_i(x) dx - C \varphi_i(0)$$

# How the Neumann condition impacts the element matrix and vector

The extra term  $C\varphi_0(0)$  affects only the element vector from the first cells since  $\varphi_0 = 0$  on all other cells.

$$\tilde{A}^{(0)} = A = \frac{1}{h} \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix}, \quad \tilde{b}^{(0)} = \begin{pmatrix} h - C \\ h \end{pmatrix} \quad (87)$$

# The finite element algorithm

The differential equation problem defines the integrals in the variational formulation.

Request these functions from the user:

```
integrand_lhs(phi, r, s, x)
boundary_lhs(phi, r, s, x)
integrand_rhs(phi, r, x)
boundary_rhs(phi, r, x)
```

Must also have a mesh with vertices, cells, and dof\_map

# Python pseudo code; the element matrix and vector

```
<Declare global matrix, global rhs: A, b>

# Loop over all cells
for e in range(len(cells)):

    # Compute element matrix and vector
    n = len(dof_map[e]) # no of dofs in this element
    h = vertices[cells[e][1]] - vertices[cells[e][0]]
    <Declare element matrix, element vector: A_e, b_e>

    # Integrate over the reference cell
    points, weights = <numerical integration rule>
    for X, w in zip(points, weights):
        phi = <basis functions + derivatives at X>
        detJ = h/2
        x = <affine mapping from X>
        for r in range(n):
            for s in range(n):
                A_e[r,s] += integrand_lhs(phi, r, s, x)*detJ*w
                b_e[r] += integrand_rhs(phi, r, x)*detJ*w

    # Add boundary terms
    for r in range(n):
        for s in range(n):
            A_e[r,s] += boundary_lhs(phi, r, s, x)*detJ*w
            b_e[r] += boundary_rhs(phi, r, x)*detJ*w
```

# Python pseudo code; boundary conditions and assembly

```
for e in range(len(cells)):
    ...

    # Incorporate essential boundary conditions
    for r in range(n):
        global_dof = dof_map[e][r]
        if global_dof in essbc_dofs:
            # dof r is subject to an essential condition
            value = essbc_docs[global_dof]
            # Symmetric modification
            b_e -= value*A_e[:,r]
            A_e[r,:] = 0
            A_e[:,r] = 0
            A_e[r,r] = 1
            b_e[r] = value

    # Assemble
    for r in range(n):
        for s in range(n):
            A[dof_map[e][r], dof_map[e][r]] += A_e[r,s]
            b[dof_map[e][r]] += b_e[r]

<solve linear system>
```

# Variational formulations in 2D and 3D

How to do integration by parts is the major difference when moving to 2D and 3D.



Rule for multi-dimensional integration by parts.

$$-\int_{\Omega} \nabla \cdot (a(\mathbf{x}) \nabla u) v \, dx = \int_{\Omega} a(\mathbf{x}) \nabla u \cdot \nabla v \, dx - \int_{\partial\Omega} a \frac{\partial u}{\partial n} v \, ds \quad (88)$$

- $\int_{\Omega}() \, dx$ : area (2D) or volume (3D) integral
- $\int_{\partial\Omega}() \, ds$ : line(2D) or surface (3D) integral
- $\partial\Omega_N$ : Neumann conditions  $-a \frac{\partial u}{\partial n} = g$
- $\partial\Omega_D$ : Dirichlet conditions  $u = u_0$
- $v \in V$  must vanish on  $\partial\Omega_D$  (in method 1)

## Example on integration by parts; problem

$$\mathbf{v} \cdot \nabla u + \alpha u = \nabla \cdot (a \nabla u) + f, \quad \mathbf{x} \in \Omega \quad (89)$$

$$u = u_0, \quad \mathbf{x} \in \partial\Omega_D \quad (90)$$

$$-a \frac{\partial u}{\partial n} = g, \quad \mathbf{x} \in \partial\Omega_N \quad (91)$$

- Known:  $a$ ,  $\alpha$ ,  $f$ ,  $u_0$ , and  $g$ .
- Second-order PDE: must have *exactly one boundary condition at each point of the boundary*

Method 1 with boundary function and  $\psi_i = 0$  on  $\partial\Omega_D$ :

$$u(\mathbf{x}) = B(\mathbf{x}) + \sum_{j \in \mathcal{I}_s} c_j \psi_j(\mathbf{x}), \quad B(\mathbf{x}) = u_0(\mathbf{x})$$

## Example on integration by parts; details (1)

Galerkin's method: multiply by  $v \in V$  and integrate over  $\Omega$ ,

$$\int_{\Omega} (\mathbf{v} \cdot \nabla u + \alpha u) v \, dx = \int_{\Omega} \nabla \cdot (a \nabla u) \, dx + \int_{\Omega} f v \, dx$$

Integrate second-order term by parts:

$$\int_{\Omega} \nabla \cdot (a \nabla u) v \, dx = - \int_{\Omega} a \nabla u \cdot \nabla v \, dx + \int_{\partial\Omega} a \frac{\partial u}{\partial n} v \, ds,$$

Resulting variational form:

$$\int_{\Omega} (\mathbf{v} \cdot \nabla u + \alpha u) v \, dx = - \int_{\Omega} a \nabla u \cdot \nabla v \, dx + \int_{\partial\Omega} a \frac{\partial u}{\partial n} v \, ds + \int_{\Omega} f v \, dx$$

## Example on integration by parts; details (2)

Note:  $v \neq 0$  only on  $\partial\Omega_N$ :

$$\int_{\partial\Omega} a \frac{\partial u}{\partial n} v \, ds = \int_{\partial\Omega_N} \underbrace{a \frac{\partial u}{\partial n}}_{-g} v \, ds = - \int_{\partial\Omega_N} g v \, ds$$

The final variational form:

$$\int_{\Omega} (\mathbf{v} \cdot \nabla u + \alpha u) v \, dx = - \int_{\Omega} a \nabla u \cdot \nabla v \, dx - \int_{\partial\Omega_N} g v \, ds + \int_{\Omega} f v \, dx$$

Or with inner product notation:

$$(\mathbf{v} \cdot \nabla u, v) + (\alpha u, v) = -(a \nabla u, \nabla v) - (g, v)_N + (f, v)$$

$(g, v)_N$ : line or surface integral over  $\partial\Omega_N$ .

## Example on integration by parts; linear system

$$u = B + \sum_{j \in \mathcal{I}_s} c_j \psi_j, \quad B = u_0$$

$$A_{i,j} = (\mathbf{v} \cdot \nabla \psi_j, \psi_i) + (\alpha \psi_j, \psi_i) + (a \nabla \psi_j, \nabla \psi_i)$$

$$b_i = (g, \psi_i)_N + (f, \psi_i) - (\mathbf{v} \cdot \nabla u_0, \psi_i) + (\alpha u_0, \psi_i) + (a \nabla u_0, \nabla \psi_i)$$

# Transformation to a reference cell in 2D/3D (1)

We want to compute an integral in the physical domain by integrating over the reference cell.

$$\int_{\Omega^{(e)}} a(\mathbf{x}) \nabla \varphi_i \cdot \nabla \varphi_j \, d\mathbf{x} \quad (92)$$

Mapping from reference to physical coordinates:

$$\mathbf{x}(\mathbf{X})$$

with Jacobian  $J$ ,

$$J_{i,j} = \frac{\partial x_j}{\partial X_i}$$

- $d\mathbf{x} \rightarrow \det J \, d\mathbf{X}$ .
- Must express  $\nabla \varphi_i$  by an expression with  $\tilde{\varphi}_r$ ,  $i = q(e, r)$ :  
 $\nabla \tilde{\varphi}_r(\mathbf{X})$

## Transformation to a reference cell in 2D/3D (2)

Can derive

$$\nabla_{\mathbf{X}} \tilde{\varphi}_r = J \cdot \nabla_{\mathbf{x}} \varphi_i$$

$$\nabla_{\mathbf{x}} \varphi_i = \nabla_{\mathbf{x}} \tilde{\varphi}_r(\mathbf{X}) = J^{-1} \cdot \nabla_{\mathbf{X}} \tilde{\varphi}_r(\mathbf{X})$$

Integral transformation from physical to reference coordinates:

$$\int_{\Omega^{(e)}} a(\mathbf{x}) \nabla_{\mathbf{x}} \varphi_i \cdot \nabla_{\mathbf{x}} \varphi_j \, d\mathbf{x} = \int_{\tilde{\Omega}_r} a(\mathbf{x}(\mathbf{X})) (J^{-1} \cdot \nabla_{\mathbf{X}} \tilde{\varphi}_r) \cdot (J^{-1} \cdot \nabla_{\mathbf{X}} \tilde{\varphi}_s) \det J \, d\mathbf{X} \quad (93)$$

# Numerical integration

Numerical integration over reference cell triangles and tetrahedra:

$$\int_{\tilde{\Omega}^r} g \, dX = \sum_{j=0}^{n-1} w_j g(\bar{\mathbf{X}}_j)$$

Module `numint.py` contains different rules:

```
>>> import numint
>>> x, w = numint.quadrature_for_triangles(num_points=3)
>>> x
[(0.16666666666666666, 0.16666666666666666),
 (0.6666666666666666, 0.16666666666666666),
 (0.16666666666666666, 0.66666666666666666)]
>>> w
[0.16666666666666666, 0.16666666666666666, 0.16666666666666666]
```

- Triangle: rules with  $n = 1, 3, 4, 7$  integrate exactly polynomials of degree 1, 2, 3, 4, resp.
- Tetrahedron: rules with  $n = 1, 4, 5, 11$  integrate exactly polynomials of degree 1, 2, 3, 4, resp.



# Time-dependent problems

- So far: used the finite element framework for discretizing in space
- What about  $u_t = u_{xx} + f$ ?
  - 1 Use finite differences in time to obtain a set of recursive spatial problems
  - 2 Solve the spatial problems by the finite element method

## Example: diffusion problem

$$\frac{\partial u}{\partial t} = \alpha \nabla^2 u + f(\mathbf{x}, t), \quad \mathbf{x} \in \Omega, t \in (0, T] \quad (94)$$

$$u(\mathbf{x}, 0) = l(\mathbf{x}), \quad \mathbf{x} \in \Omega \quad (95)$$

$$\frac{\partial u}{\partial n} = 0, \quad \mathbf{x} \in \partial\Omega, t \in (0, T] \quad (96)$$

## A Forward Euler scheme; ideas

$$[D_t^+ u = \alpha \nabla^2 u + f]^n, \quad n = 1, 2, \dots, N_t - 1 \quad (97)$$

Solving wrt  $u^{n+1}$ :

$$u^{n+1} = u^n + \Delta t (\alpha \nabla^2 u^n + f(\mathbf{x}, t_n)) \quad (98)$$

- $u^n = \sum_j c_j^n \psi_j \in V$ ,  $u^{n+1} = \sum_j c_j^{n+1} \psi_j \in V$
- Compute  $u^0$  from  $I$
- Compute  $u^{n+1}$  from  $u^n$  by solving the PDE for  $u^{n+1}$  at each time level

## A Forward Euler scheme; stages in the discretization

- $u_e(\mathbf{x}, t)$ : exact solution of the space-and time-continuous problem
- $u_e^n(\mathbf{x})$ : exact solution of time-discrete problem (after applying a finite difference scheme in time)
- $u_e^n(\mathbf{x}) \approx u^n = \sum_{j \in \mathcal{I}_s} c_j^n \psi_j =$  solution of the time- and space-discrete problem (after applying a Galerkin method in space)

$$\frac{\partial u_e}{\partial t} = \alpha \nabla^2 u_e + f(\mathbf{x}, t) \quad (99)$$

$$u_e^{n+1} = u_e^n + \Delta t (\alpha \nabla^2 u_e^n + f(\mathbf{x}, t_n)) \quad (100)$$

$$u_e^n \approx u^n = \sum_{j=0}^N c_j^n \psi_j(\mathbf{x}), \quad u_e^{n+1} \approx u^{n+1} = \sum_{j=0}^N c_j^{n+1} \psi_j(\mathbf{x})$$

$$R = u^{n+1} - u^n - \Delta t (\alpha \nabla^2 u^n + f(\mathbf{x}, t_n))$$

# A Forward Euler scheme; weighted residual (or Galerkin) principle

$$R = u^{n+1} - u^n - \Delta t (\alpha \nabla^2 u^n + f(\mathbf{x}, t_n))$$

The weighted residual principle:

$$\int_{\Omega} R w \, dx = 0, \quad \forall w \in W$$

results in

$$\int_{\Omega} [u^{n+1} - u^n - \Delta t (\alpha \nabla^2 u^n + f(\mathbf{x}, t_n))] w \, dx = 0, \quad \forall w \in W$$

Galerkin:  $W = V$ ,  $w = v$

## A Forward Euler scheme; integration by parts

Isolating the unknown  $u^{n+1}$  on the left-hand side:

$$\int_{\Omega} u^{n+1} \psi_i \, dx = \int_{\Omega} [u^n - \Delta t (\alpha \nabla^2 u^n + f(\mathbf{x}, t_n))] v \, dx$$

Integration by parts of  $\int \alpha(\nabla^2 u^n) v \, dx$ :

$$\int_{\Omega} \alpha(\nabla^2 u^n) v \, dx = - \int_{\Omega} \alpha \nabla u^n \cdot \nabla v \, dx + \underbrace{\int_{\partial\Omega} \alpha \frac{\partial u^n}{\partial n} v \, dx}_{=0 \quad \Leftarrow \quad \partial u^n / \partial n = 0}$$

Variational form:

$$\int_{\Omega} u^{n+1} v \, dx = \int_{\Omega} u^n v \, dx - \Delta t \int_{\Omega} \alpha \nabla u^n \cdot \nabla v \, dx + \Delta t \int_{\Omega} f^n v \, dx, \quad \forall v \in V \quad (101)$$

## New notation for the solution at the most recent time levels

- $u$  and  $u$ : the spatial unknown function to be computed
- $u_1$  and  $u_1$ : the spatial function at the previous time level  $t - \Delta t$
- $u_2$  and  $u_2$ : the spatial function at  $t - 2\Delta t$
- This new notation gives close correspondance between code and math

$$\int_{\Omega} uv \, dx = \int_{\Omega} u_1 v \, dx - \Delta t \int_{\Omega} \alpha \nabla u_1 \cdot \nabla v \, dx + \Delta t \int_{\Omega} f^n v \, dx \quad (102)$$

or shorter

$$(u, \psi_i) = (u_1, v) - \Delta t(\alpha \nabla u_1, \nabla v) + (f^n, v) \quad (103)$$

# Deriving the linear systems

- $u = \sum_{j=0}^N c_j \psi_j(\mathbf{x})$
- $u_1 = \sum_{j=0}^N c_{1,j} \psi_j(\mathbf{x})$
- $\forall v \in V$ : for  $v = \psi_i$ ,  $i = 0, \dots, N$

Insert these in

$$(u, \psi_i) = (u_1, \psi_i) - \Delta t (\alpha \nabla u_1, \nabla \psi_i) + (f^n, \psi_i)$$

and order terms as matrix-vector products:

$$\sum_{j=0}^N \underbrace{(\psi_i, \psi_j)}_{M_{i,j}} c_j = \sum_{j=0}^N \underbrace{(\psi_i, \psi_j)}_{M_{i,j}} c_{1,j} - \Delta t \sum_{j=0}^N \underbrace{(\nabla \psi_i, \alpha \nabla \psi_j)}_{K_{i,j}} c_{1,j} + (f^n, \psi_i), \quad i =$$

(104)



$$Mc = Mc_1 - \Delta t Kc_1 + f \quad (105)$$

$$M = \{M_{i,j}\}, \quad M_{i,j} = (\psi_i, \psi_j), \quad i, j \in \mathcal{I}_s$$

$$K = \{K_{i,j}\}, \quad K_{i,j} = (\nabla \psi_i, \alpha \nabla \psi_j), \quad i, j \in \mathcal{I}_s$$

$$f = \{(f(\mathbf{x}, t_n), \psi_i)\}_{i \in \mathcal{I}_s}$$

$$c = \{c_i\}_{i \in \mathcal{I}_s}$$

$$c_1 = \{c_{1,i}\}_{i \in \mathcal{I}_s}$$

# Computational algorithm

- ➊ Compute  $M$  and  $K$ .
- ➋ Initialize  $u^0$  by either interpolation or projection
- ➌ For  $n = 1, 2, \dots, N_t$ :
  - ➊ compute  $b = Mc_1 - \Delta t Kc_1 + f$
  - ➋ solve  $Mc = b$
  - ➌ set  $c_1 = c$

Initial condition:

- Either interpolation:  $c_{1,j} = I(\mathbf{x}_j)$  (finite elements)
- Or projection: solve  $\sum_j M_{i,j} c_{1,j} = (I, \psi_i)$ ,  $i \in \mathcal{I}_s$

# Comparing P1 elements with the finite difference method; ideas

- P1 elements in 1D
- Uniform mesh on  $[0, L]$  with cell length  $h$
- No Dirichlet conditions:  $\psi_i = \varphi_i$ ,  $i = 0, \dots, N = N_n$
- Have found formulas for  $M$  and  $K$  at the element level
- Have assembled the global matrices
- Have developed corresponding finite difference operator formulas
- $M$ :  $h[D_t^+(u + \frac{1}{6}h^2 D_x D_x u)]_i^n$
- $K$ :  $h[\alpha D_x D_x u]_i^n$

# Comparing P1 elements with the finite difference method; results

Diffusion equation with finite elements is equivalent to

$$[D_t^+(u + \frac{1}{6}h^2 D_x D_x u) = \alpha D_x D_x u + f]_i^n \quad (106)$$

Can lump the mass matrix by Trapezoidal integration and get the standard finite difference scheme

$$[D_t^+ u = \alpha D_x D_x u + f]_i^n \quad (107)$$

# Discretization in time by a Backward Euler scheme

Backward Euler scheme in time:

$$[D_t^- u = \alpha \nabla^2 u + f(\mathbf{x}, t)]^n.$$

$$u_e^n - \Delta t (\alpha \nabla^2 u_e^n + f(\mathbf{x}, t_n)) = u_e^{n-1} \quad (108)$$

$$u_e^n \approx u^n = \sum_{j=0}^N c_j^n \psi_j(\mathbf{x}), \quad u_e^{n+1} \approx u^{n+1} = \sum_{j=0}^N c_j^{n+1} \psi_j(\mathbf{x})$$

## The variational form of the time-discrete problem

$$\int_{\Omega} (u^n v + \Delta t \alpha \nabla u^n \cdot \nabla v) \, dx = \int_{\Omega} u^{n-1} v \, dx - \Delta t \int_{\Omega} f^n v \, dx, \quad \forall v \in V \quad (109)$$

or

$$(u, v) + \Delta t (\alpha \nabla u, \nabla v) = (u_1, v) + \Delta t (f^n, \psi_i) \quad (110)$$

The linear system: insert  $u = \sum_j c_j \psi_j$  and  $u_1 = \sum_j c_{1,j} \psi_j$ ,

$$(M + \Delta t \alpha K) c = M c_1 + f \quad (111)$$

Can interpret the resulting equation system as

$$[D_t^-(u + \frac{1}{6}h^2 D_x D_x u) = \alpha D_x D_x u + f]_i^n \quad (112)$$

Lumped mass matrix (by Trapezoidal integration) gives a standard finite difference method:

$$[D_t^- u = \alpha D_x D_x u + f]_i^n \quad (113)$$

# Dirichlet boundary conditions

Dirichlet condition at  $x = 0$  and Neumann condition at  $x = L$ :

$$u(\mathbf{x}, t) = u_0(\mathbf{x}, t), \quad \mathbf{x} \in \partial\Omega_D \quad (114)$$

$$-\alpha \frac{\partial}{\partial n} u(\mathbf{x}, t) = g(\mathbf{x}, t), \quad \mathbf{x} \in \partial\Omega_N \quad (115)$$

Forward Euler in time, Galerkin's method, and integration by parts:

$$\int_{\Omega} u^{n+1} v \, dx = \int_{\Omega} (u^n - \Delta t \alpha \nabla u^n \cdot \nabla v) \, dx - \Delta t \int_{\partial\Omega_N} g v \, ds, \quad \forall v \in V \quad (116)$$

Requirement:  $v = 0$  on  $\partial\Omega_D$



$$u^n(\mathbf{x}) = u_0(\mathbf{x}, t_n) + \sum_{j \in \mathcal{I}_s} c_j^n \psi_j(\mathbf{x})$$

$$\begin{aligned} \sum_{j \in \mathcal{I}_s} \left( \int_{\Omega} \psi_i \psi_j \, d\mathbf{x} \right) c_j^{n+1} = & \sum_{j \in \mathcal{I}_s} \left( \int_{\Omega} (\psi_i \psi_j - \Delta t \alpha \nabla \psi_i \cdot \nabla \psi_j) \, d\mathbf{x} \right) c_j^n - \\ & \int_{\Omega} (u_0(\mathbf{x}, t_{n+1}) - u_0(\mathbf{x}, t_n) + \Delta t \alpha \nabla u_0(\mathbf{x}, t_n) \cdot \nabla \psi_i) \, d\mathbf{x} \\ & + \Delta t \int_{\Omega} f \psi_i \, d\mathbf{x} - \Delta t \int_{\partial\Omega_N} g \psi_i \, ds, \quad i \in \mathcal{I}_s \end{aligned}$$

# Finite element basis functions

- $B(\mathbf{x}, t_n) = \sum_{j \in I_b} U_j^n \varphi_j$
- $\psi_i = \varphi_{\nu(j)}, j \in \mathcal{I}_s$
- $\nu(j), j \in \mathcal{I}_s$ , are the node numbers corresponding to all nodes without a Dirichlet condition

$$u^n = \sum_{j \in I_b} U_j^n \varphi_j + \sum_{j \in \mathcal{I}_s} c_{1,j} \varphi_{\nu(j)},$$

$$u^{n+1} = \sum_{j \in I_b} U_j^{n+1} \varphi_j + \sum_{j \in \mathcal{I}_s} c_{j} \varphi_{\nu(j)}$$

$$\begin{aligned} \sum_{j \in \mathcal{I}_s} \left( \int_{\Omega} \varphi_i \varphi_j \, dx \right) c_j &= \sum_{j \in \mathcal{I}_s} \left( \int_{\Omega} (\varphi_i \varphi_j - \Delta t \alpha \nabla \varphi_i \cdot \nabla \varphi_j) \, dx \right) c_{1,j} - \\ &\quad \sum_{j \in I_b} \int_{\Omega} \left( \varphi_i \varphi_j (U_j^{n+1} - U_j^n) + \Delta t \alpha \nabla \varphi_i \cdot \nabla \varphi_j U_j^n \right) dx \\ &\quad + \Delta t \int_{\Omega} f \varphi_i \, dx - \Delta t \int_{\partial \Omega_N} g \varphi_i \, ds, \quad i \in \mathcal{I}_s \end{aligned}$$

# Modification of the linear system; the raw system

- Drop boundary function
- Compute as if there are not Dirichlet conditions
- Modify the linear system to incorporate Dirichlet conditions
- $\mathcal{I}_s$  holds the indices of all nodes  $\{0, 1, \dots, N = N_n\}$

$$\sum_{j \in \mathcal{I}_s} \underbrace{\left( \int_{\Omega} \varphi_i \varphi_j \, dx \right)}_{M_{i,j}} c_j = \sum_{j \in \mathcal{I}_s} \left( \underbrace{\int_{\Omega} \varphi_i \varphi_j \, dx}_{M_{i,j}} - \Delta t \underbrace{\int_{\Omega} \alpha \nabla \varphi_i \cdot \nabla \varphi_j \, dx}_{K_{i,j}} \right) c_{1,j} \\ - \Delta t \underbrace{\int_{\Omega} f \varphi_i \, dx - \Delta t \int_{\partial \Omega_N} g \varphi_i \, ds}_{f_i}, \quad i \in \mathcal{I}_s$$

## Modification of the linear system; setting Dirichlet conditions

$$Mc = b, \quad b = Mc_1 - \Delta t K c_1 + \Delta t f \quad (117)$$

For each  $k$  where a Dirichlet condition applies,  $u(x_k, t_{n+1}) = U_k^{n+1}$ ,

- set row  $k$  in  $M$  to zero and 1 on the diagonal:  $M_{k,j} = 0$ ,  
 $j \in \mathcal{I}_s$ ,  $M_{k,k} = 1$
- $b_k = U_k^{n+1}$

Or apply the slightly more complicated modification which preserves symmetry of  $M$

# Modification of the linear system; Backward Euler example

Backward Euler discretization in time gives a more complicated coefficient matrix:

$$Ac = b, \quad A = M + \Delta t K, \quad b = Mc_1 + \Delta t f. \quad (118)$$

- Set row  $k$  to zero and 1 on the diagonal:  $M_{k,j} = 0, j \in \mathcal{I}_s$ ,  
 $M_{k,k} = 1$
- Set row  $k$  to zero:  $K_{k,j} = 0, j \in \mathcal{I}_s$
- $b_k = U_k^{n+1}$

Observe:  $A_{k,k} = M_{k,k} + \Delta t K_{k,k} = 1 + 0$ , so  $c_k = U_k^{n+1}$

# Analysis of the discrete equations

The diffusion equation  $u_t = \alpha u_{xx}$  allows a (Fourier) wave component

$$u = A_e^n e^{ikx}, \quad A_e = e^{-\alpha k^2 \Delta t} \quad (119)$$

Numerical schemes often allow the similar solution

$$u_q^n = A^n e^{ikx} \quad (120)$$

- $A$ : amplification factor to be computed
- How good is this  $A$  compared to the exact one?

## Handy formulas

$$[D_t^+ A^n e^{ikq\Delta x}]^n = A^n e^{ikq\Delta x} \frac{A - 1}{\Delta t},$$

$$[D_t^- A^n e^{ikq\Delta x}]^n = A^n e^{ikq\Delta x} \frac{1 - A^{-1}}{\Delta t},$$

$$[D_t A^n e^{ikq\Delta x}]^{n+\frac{1}{2}} = A^{n+\frac{1}{2}} e^{ikq\Delta x} \frac{A^{\frac{1}{2}} - A^{-\frac{1}{2}}}{\Delta t} = A^n e^{ikq\Delta x} \frac{A - 1}{\Delta t},$$

$$[D_x D_x A^n e^{ikq\Delta x}]_q = -A^n \frac{4}{\Delta x^2} \sin^2 \left( \frac{k\Delta x}{2} \right).$$

# Amplification factor for the Forward Euler method; results

Introduce  $p = k\Delta x/2$  and  $C = \alpha\Delta t/\Delta x^2$ :

$$A = 1 - 4C \frac{\sin^2 p}{1 - \underbrace{\frac{2}{3} \sin^2 p}_{\text{from } M}}$$

(See notes for details)

Stability:  $|A| \leq 1$ :

$$C \leq \frac{1}{6} \quad \Rightarrow \quad \Delta t \leq \frac{\Delta x^2}{6\alpha} \quad (121)$$

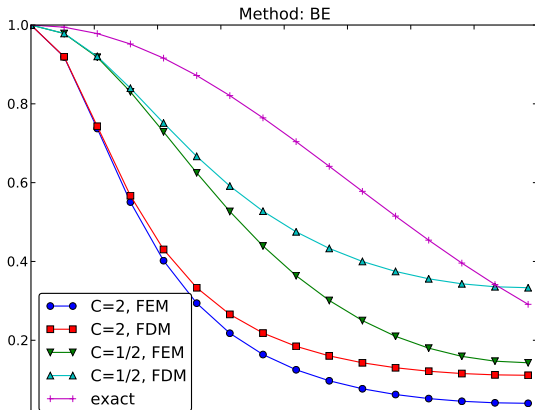
Finite differences:  $C \leq \frac{1}{2}$ , so finite elements give a *stricter* stability criterion for this PDE!



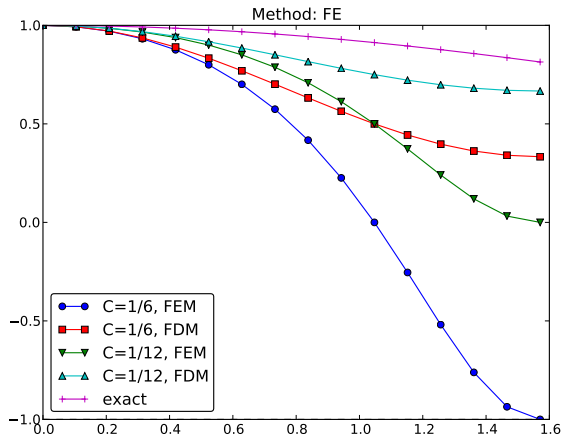
# Amplification factor for the Backward Euler method; results

Coarse meshes:

$$A = \left( 1 + 4C \frac{\sin^2 p}{1 + \frac{2}{3} \sin^2 p} \right)^{-1} \quad (\text{unconditionally stable})$$



# Amplification factors for smaller time steps; Forward Euler



# Amplification factors for smaller time steps; Backward Euler

